Assembler Code Description:

1. Data Structures Used

To implement the assembler, we had to make use of one principal data structure, which is the hash table. It was used for two primary purposes:

- To keep track of the symbols used in the code to map them to the memory addresses containing their values.
- To keep track of the labels used in the code to retrieve them later similarly.

We decided to use hash tables because of their low time complexity in insertion and retrieval: Both O(1), which will be efficient since we will be using them inside loops throughout the program. It is also better if we have a long code where we wouldn't want to waste time in these basic operations.

2. Functions

We chose to divide this code into multiple functions in order to have a clear separation between the parts that should be handled separately and to have better code readability.

- initializeTables()

  → This function is used to initially create the hash tables that are going to be used in the program, and it allocated the memory according to the predefined hash size.
  The function takes no input arguments since it uses the global hash table variables.

- hashfunction()

  → Used to assign to each string (key) an integer value which is later used as the cell's index in which the element will be inserted in the hash table.
  It takes as input argument the key, which is going to be used to assign a hash code to it using the loop:

  ```
  for (hashed=0; *key!='\0'; key++) {
          hashed = *key + 28*hashed;
  }
  ```

  Where:

  hashed: the hash code generated
  *key: ASCII value of a particular character in the key

We thought it would be best to calculate the hash code this way to ensure a minimum number of collisions since the sum of ASCII values of each string are unlikely to match with those of another string.

- find()

  → It is used to find a particular element in the hash table. It takes as an argument the key it needs to look for and returns the index's value where it was found. It also takes the hash table where it needs to look up the key since we are using this function to find both labels and symbols in the labelable and the symbolTable.

- insertElement()

  → It is used to insert a key and its value in a particular hash table. It takes as an argument the key to be inserted, the value that it needs to be mapped to, and the hash table where it will be inserted since it will be used to insert both labels and symbols in the labelTable or symbolTable.

  This function handles collisions using quadratic probing. Whenever a collision is encountered, a new index is computed using the formula:

  $$Index = (loopindex^2 + Index) \% HASHSIZE$$

  Where:

  - Index: is the new computed index
  - loopindex: an integer that gets incremented by 1 (1,2,3…)
  - HASHSIZE: hashTable size, which we chose to set to 100 for all the tables.

  We chose to handle collisions this way since linear probing would take a longer time, so quadratic probing seemed to be a better choice. We avoided using chaining since we assumed it would be a bit too complex for our project. These are the three collision handling techniques our team members are familiar with, and so based on these criteria, we went for quadratic probing.

- initializeData()

  → It is used to handle the data part of the program. It takes care of inserting the symbols encountered in this part in the corresponding symbolTable hash table and maps them to their memory addresses. It also halts the code's execution if an unexpected opcode is found, i.e.,

any opcode different from DEF, which is used to define symbols in the data part, or when the separator is found.
This function takes as input arguments both the input file and the output file, and it prints the corresponding machine code of the data part in the output file.

- initializeProgram()
  → It is used to handle the program part. It takes care of inserting any labels it encounters in the corresponding labelTable hash table and maps them to the referencing location. It also contains an if-else statement that takes care of swapping the assembly code opcodes with their machine code equivalent.
  After each swapping operation, the function prints the corresponding machine code in the output file. It also halts the execution if an unexpected error is found or when the separator is found.
  The function takes as input the input file and the output file.

- initializeInput()
  → This function takes care of copy-pasting the input data part from the input file to the output file since no modifications are needed, and only integer values are written in this part.
  The function takes as input both files as well.

- initializelabels()
  →This function takes care of initializing the labels in the label table. It reads the whole input file line by line looking for instructions with opcode "ASG" and gets the label from the first operand, then it associates it to a line value.

3. How to Run the Code

This assembler takes as input a file that contains a program written in assembly code, then it converts it to machine code and writes the output machine code to another output file, which the interpreter will later use.
To run the code, a file containing the assembly code, named "assembly.txt" needs to be created and filled in the exact location where the assembler code is stored.
The code can be run on any IDE provided that a GCC compiler is installed.

4. Sample Input and Output

1. Simple Code

- Assembly code in input file:

```
DEF num1 0000
DEF num2 0000
DEF num3 0000
DEF AVG  0000
+99 9999 9999
READ num1 0000
READ num2 0000
READ num3 0000
MOV num1 0000
ADD num2 num2
MOV num2 0000
ADD num3 num3
MOV 0003 0000
DIV num3 AVG
PRNT AVG 0000
HALT 0000 0000
+99 9999 9999
+0 0000 0060
+0 0000 0070
+0 0000 0080
```

- Machine code in output file:

```
+000000000
+000010000
+000020000
+000030000
+9999999999
+7300000000
+7300010000
+7300020000
+0300000000
+1100010001
+0300010000
+1100020002
+0000030000
-2100020003
-7300030000
+9000000000
+9999999999
+000000060
+000000070
```

```
+000000080
```

2. Medium Complexity Code

- Assembly code in input file:

```
DEF num1 0000
+99 9999 9999
READ num1 0000
SLT num1 END
ASG LOOP 0000
PRNT num1 0000
MOV 0001  0000
SUB num1 num1
MOV 0000 0000
GTE num1 LOOP
ASG END 0000
HALT 0000 0000
+99 9999 9999
00 0000 0005
```

- Machine code in output file:

```
+000000000
+9999999999
+7300000000
-5100000007
-7300000000
+0000010000
-1100000000
+0000000000
+5100000002
+9000000000
+9999999999
0000000005
```

3. High Complexity Code

- Assembly code in input file:

```
DEF num1 0000
DEF FACT 0001
DEF PRD 0000
+99 9999 9999
```

```
READ num1 0000
SLT num1 END
ASG LOOP 0000
PRNT num1 0000
EQU num1 ZERO
MOV num1 PRD
ASG LOP2 0000
MOV PRD 0000
MUL FACT FACT
MOV 0001 0000
SUB PRD PRD
GTE PRD LOP2
ASG ZERO 0000
PRNT FACT 0000
MOV 0001 FACT
MOV 0001  0000
SUB num1 num1
MOV 0000 0000
GTE num1 LOOP
ASG END 0000
HALT 0000 0000
+99 9999 9999
+00 0000 0005
```

- Machine code in output file:

```
+000000000
+000010001
+000020000
+9999999999
+7300000000
-5100000016
-7300000000
+4100000010
+0100000002
+0300020000
+2100010001
+0000010000
-1100020002
+5100020005
-7300010000
+0200010001
+0000010000
-1100000000
```

```
+0000000000
+5100000002
+9000000000
+9999999999
+0000000005
```