Homework 2

I- Purpose of the Project:

Creating an Assembly program that takes in the string stream of a post-fix expression, evaluates it, and output the result as a hexadecimal number.

II- The General Algorithm of the Project:

The project relies principally on extracting each token individually from the expression, converting it to a numerical decimal equivalent, and pushing it to the stack. Upon reaching the end of the stack, the output is converted to a hexadecimal form and back to a stream of characters for printing on the console.

III- Overview of the Project’s structure:

The project incorporates the following subroutines as ordered in the project’s implementation:

1- main_reader

2- operand_reader

3- Add_Operation

4- Multiply

5- Divide

6- XOR_Operation

7- result_converter

8- AND_Operation

9- OR_Operation

10- process_digit

11- process_letter

12- operand_pusher

13- one_operand_input

14- global_printer

15- letter_printer

16- int_printer

In addition, there are various Jumper sub-routines that act as intermediaries between the aforementioned core sub-routines due to the jump size limitations.

IV- A brief summary on the functionality of each sub-routine:

1- main_reader:

The main driver sub-routine. In this sub-routine, we compare the input characters to those utilized to represent each respective operation and upon receiving an affirmative result that we jump to the suitable sub-routine for operation handling.

2- operand_reader:

This sub-routine is reached if the input is an alphabetical or a digital character. Since the hexadecimal notation includes characters in its representation, we compare each input character with "**:**" and upon the comparison's result that we jump to either **process_digit** or **process_letter subroutine**.

3- Add_Operation:

In this operation, 2 operands are popped from the stack and added. The result is pushed back to the stack.

4- Multiply:

In this operation, 2 operands are popped from the stack and multiplied. The result is pushed back to the stack.

5- Divide:

In this operation, 2 operands are popped from the stack and divided. While the quotient is pushed back to the stack, the remainder is ignored.

6- XOR_Operation:

In this operation, 2 operands are popped from the stack and have the XOR operation applied on them. The result is pushed back to the stack.

7- result_converter:

Upon reaching the end of the character stream, we print the output. Due to the assumption that an output would at most occupy 16 bits, we pop the last element in the stack, divide it thrice, and push the remainder of each division operation to the stack. Another division to obtain the fourth digit is unwarranted since the quotient of the third division operation will be remainder of the fourth one. Hence, we push it as well. Upon this action that we jump to the **global_printer** sub-routine.

8- AND_Operation:

In this operation, 2 operands are popped from the stack and have the AND operation applied on them. The result is pushed back to the stack.

9- OR_Operation:

In this operation, 2 operands are popped from the stack and have the OR operation applied on them. The result is pushed back to the stack.

10- process_digit:

Upon receiving a digital character from the **operand_reader's** earlier call, the character is converted to a digit by subtracting the character "**0**" from it. Since the output of the multiplication of 2 double-word-size registers is stored in the registers DX:AX, the digit is stored in a buffer variable called **temp**. Since a token might include multiple characters, we multiply the stored aggregate value of all the previously read characters by 16 and sum the result with the digit stored in **temp** to form an update value of the token. The process of multiplication and addition continue until a space character is encountered.

11- process_letter:

Upo Upon receiving a digital character from the **operand_reader's** earlier call, the character is converted to a digit by subtracting the character "**7**" from it. Since the output of the multiplication of 2 double-word-size registers is stored in the registers DX:AX, the digit is stored in a buffer variable called **temp**. Since a token might include multiple characters, we multiply the stored aggregate value of all the previously read characters by 16 and sum the result with the digit stored in **temp** to form an update value of the token. The process of multiplication and addition continue until a space character is encountered.

12- operand_pusher:

Pushes the operand (decimal value) to the stack after encountering a space character at the input char sequence.

13- global_printer: Main operation that initiates the printing actions of the result. After converting the decimal value of the result of the postfix expression to its hexadecimal value, printing actions starts. Since we decided to print the result character by character, we evaluate every digit of the hexadecimal result and decide whether it corresponds to a letter or number in hexadecimal base. After deciding the corresponding digit in hexadecimal base we jump to sub-printing operations which is based on the digit's value. (If digit's value is bigger than 9 program jumps to printing letter and jumps to printing number (int) otherwise.)

14- letter_printer:

Adds 55 (since it corresponds to a letter in ASCII table) to its value in order to calculate the corresponding ASCII character of the value of the digit in hexadecimal base and prints the processed digit.

15- int_printer:

Adds 48 (since it corresponds to a number (integer) in ASCII table) to its value in order to calculate the corresponding ASCII character of the value of the digit in hexadecimal base and prints the processed digit.

16- one_operand_input:

Deals with the one operand input edge case. Additional to the normal inputs this instruction checks whether there is an end of line character at the end of an operand. If there is one, process the given operand as a full postfix expression.