GROUP MEMBERS:
MISRA YAVUZ, 2016400135
ALGI KANAR, 20161400123

PROBLEM DESCRIPTION

Problem: Writing an A86 assembly program which takes postfix expressions written hexadecimally and after the necessary calculations, displaying the result of the expression again hexadecimally.

In detail, there are several operations need to be implemented in the calculation process which can be listed as addition, multiplication, integer division, bitwise xor, bitwise and, bitwise or. Even though these mentioned operations have similar writing principles, they could bare slight changes from one and other. We apply certain steps in our code taking account of these changes, 16 bit restriction in values and as well as possible adjacently written values.

PROBLEM SOLUTION

The plausible way to solve this task is to separate it into 3 main parts which can be named as processing the input, evaluating the expression and giving the output. Of course, these parts need to be subdivided for clear understanding.

Processing the input starts with READ_CHAR as the name suggest reading the characters. Nevertheless, we need to acknowledge the value have taken into the AL register is the ASCII value of the character. In order to proceed in calculation, we need to change this value to what we saw in expression in terms of hexadecimal. The conversion is done for both between 0-9 and above 9 giving the values between A-F. The crucial part is that we don't know in beforehand whether there are any adjacently written values in expression; therefore, we need to keep that information somehow and warn the process before doing unwanted adjacent value calculation. As we see below the label of MERGE, we are checking CL counter to obtain this information, according to that we rather proceed the combining or go back to reading. If we proceed in combining part, we use the DL register to store our previous value at first in BETWEEN label after the second adjacent value (if it comes) we are going to have the info from merge itself since we move AX to DX at last row before going back to reading. In precise, we are assigning current value to BX and then multiplying the previous value by casting it to AX from DX in order to shift it (meanwhile we are giving the temporary value of 010H to DX in order to do this.) and add them gracefully in terms of hexadecimals.

Before coming to evaluation concept, we need to talk about the essential component which is the default stack. We take our values from stack for evaluation. Then, how we fill stack? TOPUSH label moves in for this part. Space character signifies it's time to push the value into the stack. Yet there comes a problem in one of the scenarios. If we have two operands consecutively divided by space like in this example: 2 3 + 4 5 + * 2 / after the operand if we don't take precaution it will push the space value during the upcoming processes. Therefore, we use CH register to dodge from this issue. For all the calculations at the end we do CH,1H moving to warn the push label.

Now we can talk about what we specifically do in evaluations. In EVALPLUS we are getting two values from stack, add them and push them back in stack. EVALMUL also use the same principal, but we need to be careful since multiplication only takes one parameter. In division there is slight change in implementation because of the division's own nature. In EVALDIV, as we pop

from the stack, we care for the arrangement and put the first popped value to CX, where we keep the divider. Then after getting divided from the stack and evaluating division, we push the quotient to the stack and simply ignore the remainder since we are doing integer division. In cases of bitwise comparisons EVALAND, EVALOR, EVALXOR; there is not much to do. We pop top 2 values from the stack, evaluate them according to the operation symbol. To evaluate, we used A86's existing AND, OR and XOR commands. After evaluating, we again push the result to the stack and jump back to READ_CHAR.

When READ_CHAR encounters the carriage return character, we understand that input has come to an end and we jump to END_OF_IN. For jumping that many lines, we needed an intermediary step and thereby used the JUMPSTEP label. Beginning to END_OF_IN, we get the result from stack to AX. Now that the stack is empty, we put a key value(021H, arbitrary chosen) to the bottom of the stack. This key value will later help us to understand when does the stack become empty.

Now comes the converting part for output. In OUT_CONV, we convert the solution(adjacent hexadecimal digits) into single hexadecimal digits since we will print characters 1 by 1. For doing this, we divide by the value by hexadecimal 10H(copied to CX earlier) and push the remainder to the stack and repeat this until the quotient becomes zero. After separating each digit, we need to convert these hexadecimal values to according ASCII values. In TO_POP, we take the top value from stack, check its value. If it is less than 9, we jump to TO_DIGIT and add the ASCII value of '0'. If it is not less than 9, we understand that it is in the range of A to F. Then, we add 037H to convert these to their ASCII characters. At each ending of this process, we jump to PRINT_OUT and print the last converted character. Then, we repeat this converting and printing process until we reach the end of the stack. To understand that we did, we compare with our key value 021H. If that is the case, there is nothing left in the stack and everything is printed out. Taking into consideration of everything, we jump to THE_END and terminate the program.

CONCLUSION

Our program basically consists of 3 main parts: taking the input, evaluating the expression and printing the output. To implement taking input and printing output, we developed some processes to take care of multiple digit numbers and to convert ASCII values to hexadecimal values(and vice versa). Handling the conversions, remains the evaluation part. Required that we evaluate postfix expressions, we implemented our evaluating labels using stack operations. Ultimately, our program is capable of evaluating postfix expressions(assumed that all values will be 16 bit values).