Assignment 2: RPN Calculator

Assignment Description

You are to write a simple calculator for unlimited-precision unsigned integers.

Your code will be written **entirely in assembly language**.
No C code is allowed, although you can use the C standard library functions mentioned in the list below by **linking** with the C standard library.

Reverse Polish notation (RPN) is a mathematical notation in which every operator follows all its operands, for example "3 + 4" would be presented as **"3 4 +"**.For simplicity, each operator will appear on a separate line of input. Input and output operands are to be in **hexadecimal** representation.

Your program should prompt ‘calc: ‘ and wait for input. Each number or operator is entered in a separate line. For example, to do the calculation “0x7A+9” a user should type:

```
calc: 7A
calc: 09
calc: +
```

Operations are performed as is standard for an RPN calculator: any input number is pushed onto an **operand stack**. Each operation is performed on operands which are popped from the operand stack. The result, if any, is pushed onto the operand stack. The output should contain no leading zeroes, but the input may have some leading zeroes.

Note: you may **not** use the 80X86 machine stack (with the ESP stack pointer) as an operand stack, and must implement a **separate** operand stack inside a dynamically allocated array, of size 5 by default. In order to change it to a different number, the user enter a command-line argument, the opereand stack size in hexadecimal digits. Your program should work correctly with any operand stack size greater than 2.

You should print out **"Error: Operand Stack Overflow"** if the calculation attempts to push operands onto the operand stack and there is no free space on the operand stack.
You should print out **"Error: Insufficient Number of Arguments on Stack"** if an operation attempts to pop an empty stack. In any case of error, your program must return the stack to its previous state (as it was before the failed action). Your program should also count the number of operations (both successful and unsuccessful) performed. This is the return valued which returned to function main. The size of the operands is unbounded, except by the size of available heap space on your virtual memory.

The required operations

The operations to be supported by your calculator are:

- **'q' – quit**
- **'+' – unsigned addition**
  pop two operands from operand stack, and push the result, their sum
- **'p' – pop-and-print**
  pop one operand from the operand stack, and print its value to stdout
- **'d' – duplicate**
  push a copy of the top of the operand stack onto the top of the operand stack
- **'&' - bitwise AND**, X&Y with X being the top of operand stack and Y the element next to x in the operand stack.
  pop two operands from the operand stack, and push the result.
- **'|' - bitwise OR**, X|Y with X being the top of operand stack and Y the element next to x in the operand stack.
  pop two operands from the operand stack, and push the result.
- **'n' – number of hexadecimal digits**
  pop one operand from the operand stack, and push one result.
- **'*' – unsigned multiplication** (bonus item*)
  pop two operands from operand stack, and push the result, their product. You are not allowed to implement x*y as x+x+x+…+x, y times.

Assumptions

- You may assume that the input is correct (i.e. numbers in upper-case hexa, no illegal characters)
- The stack size is at most 255 (0xFF)
- Each input line is no more than 80 characters in length

Debug option

Your program should allow **"-d" command line argument**, which means a **debug option**. When "-d" option is set, you should print out to stderr various debugging messages (as a minimum, print out every number read from the user, and every result pushed onto the operand stack). This part would not be checked via automatic tests, so there is no predefined exact format of debugging messages.

Code Requirements

Provide a single assembly language file called calc.s.
Calculator functions, as well as input and output functions, must be programmed as procedures (subroutines, or functions) to maintain modularity. You may use gets() or fgets() C standard library functions for user input, and printf() C standard library function to print out calculated results or error messages. In "main" call myCalc(), which is your primary procedure. When user enters "q", your program

should exit, by having myCalc() return the total number of operations performed, not including the 'q' operation, to the "main" code, which should print out that number before exiting (in hexa).
You may use 'stdin' C standard library variable (note that 'FILE * stdin' is variable and not constant; so if you use fgets with stdin, you should push content of stdin as argument into stack - push dword [stdin])

You may call only the following C standard library functions **from your assembly language code**:

- gets, getchar, fgets, printf, fprintf, fflush, malloc, calloc, free

If you use the above functions, you should declare them as extern, as follows (declarations can be as below, or at the beginning of the file):

```
section .text
  align 16
  global main
  extern printf
  extern fprintf
  extern fflush
  extern malloc
  extern calloc
  extern free
  extern gets
  extern getchar
  extern fgets
main:
```

- Declare a label "main:" and "global main" in your assembly program
- Declare C library functions as extern, so you will be able to call them
- Note: there is no need for a C file! The gcc linker will link external C standard library functions object code to your object code
- Compile and link your assembly file calc.s as follows:

```
nasm -f elf calc.s -o calc.o
gcc -m32 -Wall -g calc.o -o calc
```

Run example

An example of user input and program output appears below.
Comments (which will not appear in input or output) are preceded by ";".

```
\> ./calc A     ; user starts the program with stack size 10
calc: 9         ; user enters a number
```

```
calc: 1         ; user enters a number
calc: +         ; user enters “addition” operator
                ; Ox9+0x1 =0xA
                ; Ox9 and 0x1 are popped, 0xA is pushed
calc: d         ; user enters “duplicate” operator, OxA is duplicated
calc: p         ; user enters pop-and-print-operator, OxA is popped and printed
A
calc: +         ; user enters “addition” operator, but there is not enough numbers in stack
Error: Insufficient Number of Arguments on Stack
calc: FE        ; user inputs a number
calc: n         ; user enters "number of hexadecimal digits” operator
                ; OxFE is popped and is used as an argument
                ; OxFE has length 2, so the number 2 is pushed
calc: d         ; user enters “duplicate” operator, Ox2 is duplicated
calc: p         ; user enters pop-and-print-operator, Ox2 is popped and printed
2
calc: &         ; user enters "X bitwise AND Y" operation
                ; X=0x2, Y=0xA, X&Y=0x2
                ; 0x2 and OxA are popped, 0x2 is pushed
calc: 39        ; user enters a number
calc: |         ; user enters "X bitwise OR Y" operation
                ; X=0x39, Y=0x2, X|Y=0x3B
                ; 0x39 and 0x2 are popped, 0x3B is pushed
calc: AB        ; user inputs a number
calc: *         ; user enters "multiplication" operator
                ; X=0xAB, Y=0x3B, X*Y=0x2769
                ; 0xAB and 0x3B are popped, 0x2769 is pushed
calc: p         ; 0x2769 is popped and printed
2769
calc: 1         ; user enters a number
calc: 2         ; user enters a number
calc: 3         ; user enters a number
calc: 4         ; user enters a number
calc: 5         ; user enters a number
calc: 6         ; user enters a number
calc: 7         ; user enters a number
calc: 8         ; user enters a number
calc: 9         ; user enters a number
calc: A         ; user enters a number
calc: B         ; user enters a number, but the stack is full
Error: Operand Stack Overflow
calc: q         ; quit calculator
B               ; number of operations (successful and not successful) is printed from main function
```

```
                ; you do not count ‘q’ operation
                ; your program should exit
```

Submission Instructions

Submit a single zip file, **ID1_ID2.zip** , includes a single assembly file calc.s . Do not add new directory structure to the zip file! Make sure you follow the coding and submission instructions correctly (print exactly as requested). **Submissions which deviate from these instructions will not be graded!**

Recommended Implementation of Unlimited Precision

In order to support **unlimited precision numbers**, each operand in the operand stack is stored as a linked list of bytes. A linked list is implemented as follows. You should, conceptually, have a struct consisting of a single byte of data and of a pointer to the next byte. Since there are no types in assembly language, any memory block of the required size (5 bytes in this case: 4 bytes for the pointer, one byte for the data) can be seen as an element of this type. Use **malloc()** or **calloc()** C standard library functions for dynamic memory allocation on demand. Ensure that you **free memory** when numbers are popped from the operand stack to avoid memory leaks.

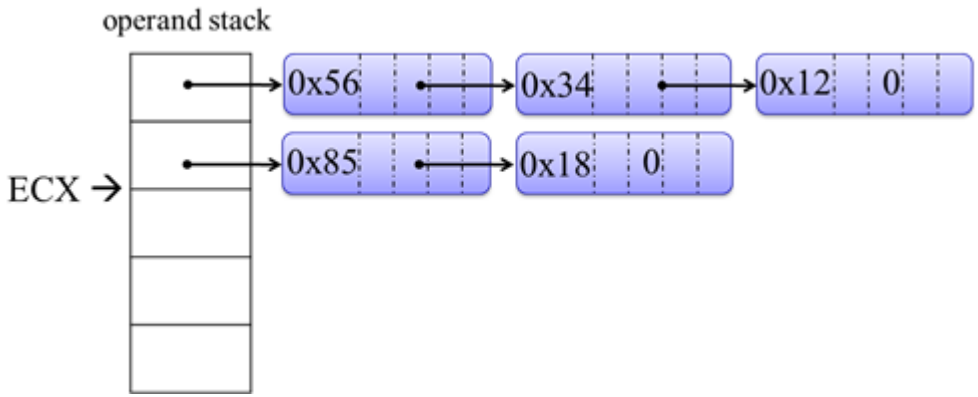
We recommend storing bytes of a number from right to left, so that the implementation of operations would be easier. If the operation you execute results in a carry from the most significant byte of the number, additional bytes must be allocated to store the result. The operand stack is best implemented as an array of pointers - each pointing to the first element of the list representing the number, or null (a null pointer has value 0). The operand stack size should be 5 pointers (5*4=20 bytes) by default.

<u>Example:</u>
0x7D12AF could be represented by the following linked list:

Suppose you insert the following numbers:

calc: 123456
calc: 1885

Then, your operand stack should be as follows:



Suppose you insert an addition action:

calc: +

Then, after the execution of the addition, your operand stack should be as follows: