Assignment 2: RPN Calculator

Assignment Description

Note the fixes in the assignment:

- when get Y>200 print "wrong Y value"
- you may use fprintf if you find it usefull
- print number of actions in hexa, there is no decimal printing in this assignment
- you may prefer use 'int getchar()' C standard library function, it is allowed
- 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]
- numbers would contain only upper case letters
- "main( )" is function and thus being called using the C calling convention. This means that before the first instruction of the function main( ) is executed the top of stack is the RETURN ADDRESS. The appropriate practical session presentation does not show this, this will be fixed in ps slides.
- in Run Example the printing error should be "Error: Insufficient Number of Arguments on Stack". Fixed.

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 enter a number “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), and must implement a **separate** oerand stack inside a static array, allocated in (e.g.) .bss. **The operand stack size is 5 slots**, specified such that, in order to change it to a different number, only one line of your code should be modified (hint: use EQU).

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 one 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
- **'^' - X*2^Y**, with X being the top of operand stack and Y the element next to x in the operand stack. If Y>200 this is considered an error, in which case you should print out an error message and leave the operand stack unaffected.
  pop two operands from the operand stack, and push one result
- **'v' – X*2^(-Y)**, with X and Y as above. This number may be not an integer. You are required to truncate the fraction part of it and keep only the integer part.
  pop two operands from the operand stack, and push one result

- **‘n’ – number of '1' bits**
  pop one operand from the operand stack, and push one result
- **‘sr’ – square root** (bonus item*)
  pop one operand from the operand stack, and push one result (only the integer part)

Assumptions

- You may assume that the input is correct (i.e. numbers in hexa, no illegal characters)
- 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 to the "main" code, which should print out that number before exiting.

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

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

If you use the above functions, you should start your code section as follows:

```
section .text
align 16
   global main
   extern printf
   extern fflush
   extern malloc
   extern calloc
   extern free
```

```
    extern gets
    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: 09        ; user enters a number
calc: 1         ; user enters a number
calc: +         ; user enters "addition" operator
                ; 0x9+0x01=0x0A
                ; 0x9 and 0x01 are popped, 0x0A is pushed
calc: d         ; user enters "duplicate" operator, 0x0A is duplicated
calc: p         ; user enters pop-and-print-operator, 0x0A 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 '1' bits" operator
                ; 0xFE is popped and is used as an argument
                ; 0xFE=11111110 contains 7 '1' bits, so the number 7 is pushed
calc: ^         ; user enters "X*2^Y" operation
                ; X=7, Y=0x0A, 7*2^A=0x1C00
                ; 7 and 0x0A are popped, 0x1C00 is pushed
calc: sr        ; user enters "square root" operator
                ; √0x1C 00=0x54
                ; 0x1C00 is popped, 0x54 is pushed
calc: p         ; 0x54 is popped and printed
54
calc: q         ; quit calculator
8               ; number of operations (successful and not successful) is printed
                ; you do not count 'quit' operation
                ; from main function
                ; 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 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 in this case: 4 for the pointer, one for the byte of data) can be seen as an element of this type. Use **malloc()** C standard library function for dynamic memory allocation on demand. Ensure that you **free memory** when numbers are popped from the operand stack to avoid memory leaks.

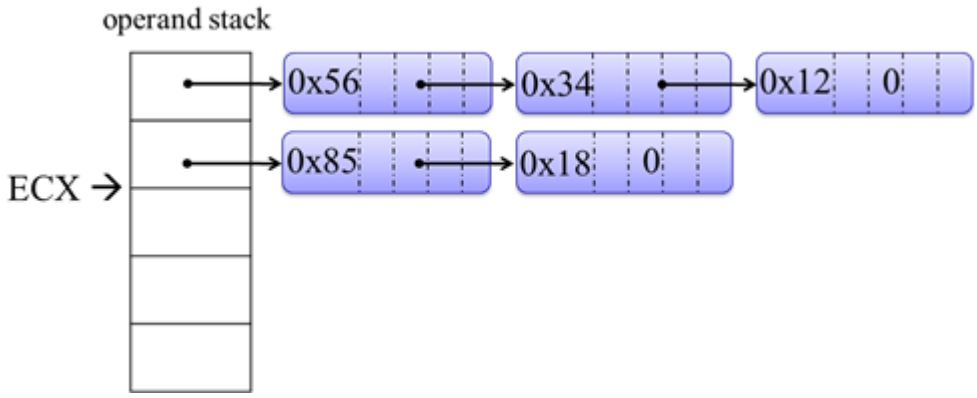
We recommend storing bytes of a number from right to left, so that the implementation of operations is be easier. If an operation results in a carry from the most significant byte of the number, additional bytes must be allocated to store the results. The operand stack is best implemented as an array 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).

Example:
0x7D12AF could be represented by the following linked list:

Suppose you insert the following numbers:

calc: 123456
calc: 1885

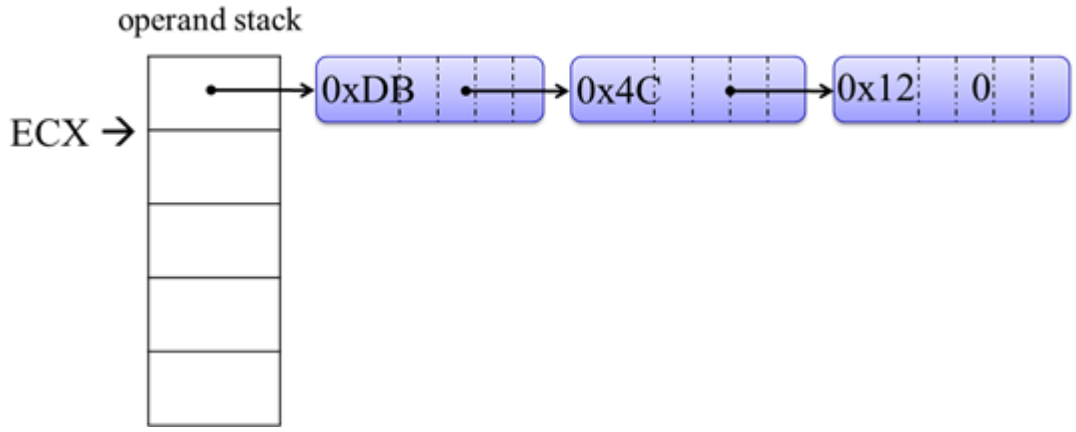
Then, your operand stack should be as follows:



Suppose you insert an addition action:

calc: +

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



Square Root – bonus item – how to calculate

How to calculate root square manually (in decimal presentation)

Let's examine the following example: $\sqrt{138385}$

1. divide the number to sections, from right to left, of two digits each section. Note that the leftmost section may contain only a single digit if the number of digits is odd
   In the example above:
   we get three sections: section 1= 13, section 2 = 83, and section 3 = 85
2. find maximal x so that x^2 < section 1
   In the example above:
   $1^2 = 1 < 13$
   $2^2 = 4 < 13$
   $3^2 = 9 < 13$
   $4^2 = 16 > 13$
   => x = 3
   => set intermediate result IR = 3
3. calculate section 1 reminder: section 1 - x^2
   compose section 2' = section 1 reminder | section 2
   In the example above:
   $x^2 = 3^2 = 9$
   section 1 reminder = $13 - 3^2 = 4$
   section 2' = 483
4. double current intermediate result: IR' = IR * 2
   find maximal x so that IR'x * x < section 2'
   In the example above:
   IR' = 3*2 = 6
   61 * 1 = 61 < 483
   62 * 2 = 124 < 483
   63 * 3 = 189 < 483
   ...
   67 * 7 = 469 < 483
   68 * 8 = 554 > 483

=> x = 7

=> set intermediate result IR = 37

5. calculate section 2 reminder: IR'x * x - section 2
   compose section 3' = section 2 reminder | section 3
   In the example above:
   IR'x * x = 67 * 7 = 469
   section 2 reminder = 469 – 483 = 14
   section 3' = 1485
6. **Execute step 4 and 5 for section 3', and for each section till the end of your input number.**
7. The reminder of the **last section** is the remainder of the square root calculation.
   In the example above:
   IR' = 37*2 = 74
   741 * 1 = 741 < 1485
   742 * 2 = 1484 < 1485
   743 * 3 = 2229 > 1485
   => x = 2
   => set final result = 372
   section 3 reminder = section 3' – 74x * x = 1485 - 742 * 2 = 1485 – 1484 = 1
   => set result reminder = 1

run example:

```
calc: 21C91        ; 21C91 hexa is 138385 decimal
calc: sr           ; result is 147 (is 372 decimal), reminder is 1
```