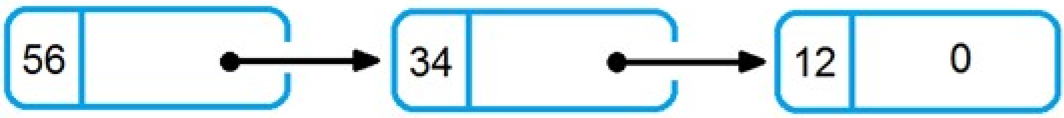


Computer Architecture and System Programming Laboratory - 2016/Spring

www.cs.bgu.ac.il |

Assignment 2: RPN Unlimited Precision BCD Calculator

Assignment Description

You are to write a simple RPN calculator for unlimited-precision unsigned integers, represented in **Binary Coded Decimal (BCD)**.

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

Each number or operator is entered in a separate line. For example, to enter a number 73 and then 80 the user should type:

73 (will saved as 01110011, **Notice**: the hexadecimal representation of the actual bits is 0x73)
80 (will saved as 10000000, **Notice**: the hexadecimal representation of the actual bits is 0x80)

As shown above, in BCD the ASCII input is read such that each character is a decimal digit, and each is then represented internally as a separate nibble (4 bits) quantity. The nibbles are then "packed" into bytes (2 nibbles, i.e. 2 BCD digits, per byte). This representation is somewhat wasteful, but simplifies the representation of numbers entered in decimal, because there is no need to do a full "decimal to binary" conversion of the entire number.

Operations are performed as is standard for an RPN calculator: any input number is pushed onto an **operand** stack, represented as an array (**not** the 80X86 machine stack), and each operation is performed on operands which are taken (and removed) from the 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).

The operand stack size should be 5, specified such that in order to change it to a different number only one line of code should be modified (hint: use EQU). You should print out "**Error: Operand**

Stack Overflow" if the calculation attempts to push too many operands onto the stack, and "**Error: Insufficient Number of Arguments on Stack**" if an operation attempts to pop an empty stack. Your program should also count the number of operations (+, p,d,&) **successfully** performed. Number size is not bounded, except by the size of available memory.

The following section suggests a recommended implementation. You **must** use a linked list of "bytes" as shown below, but the actual implementation of the linked list (e.g. whether highest byte is first or last, singly connected or double connected) is up to you.

Implementation of Unlimited Precision

In order to support **unlimited precision**, each operand in the operand stack stores a linked list (of bytes) for each operand. A linked list (of bytes, in this case) is implemented as follows. You should, conceptually, have a "type" consisting of a pointer to next, and a byte of data. Since there are no types in assembly language, any memory block of the requisite size (5 in this case: 4 for the pointer, one for the byte of data) can be seen as an element of this type. To make sure the memory blocks are free before use, you should allocate them from free memory, using **malloc()**, just as you would do in C.

If an operation results in a carry from the most significant byte, 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 still be 5.

Example:
123456 could be represented by the following linked list:

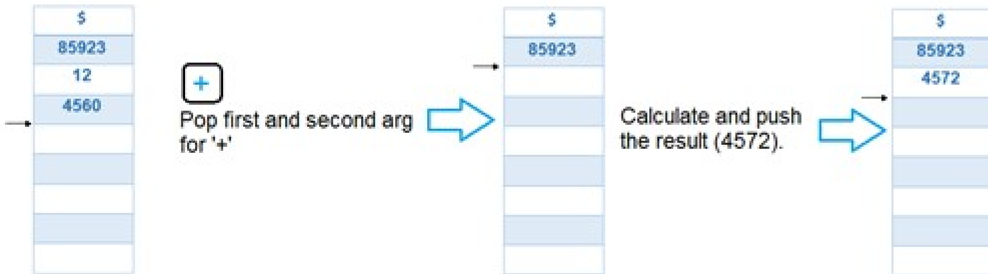
The required operations

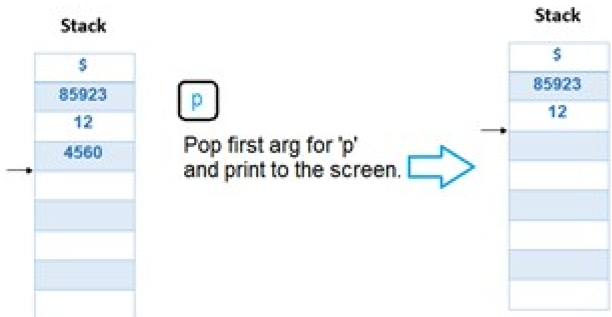
The operations to be supported by your calculator are:

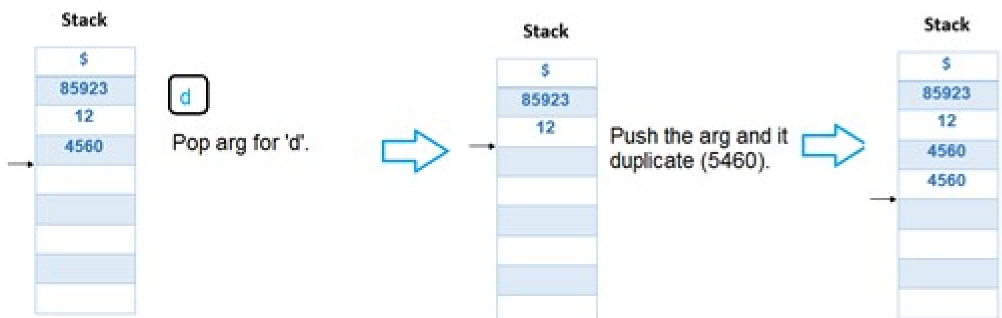
- Quit (q)
- Addition (unsigned) (+)
- Pop-and-print (p)
- Duplicate (d)
- Bitwise AND (&), on the actual bits (see example below)

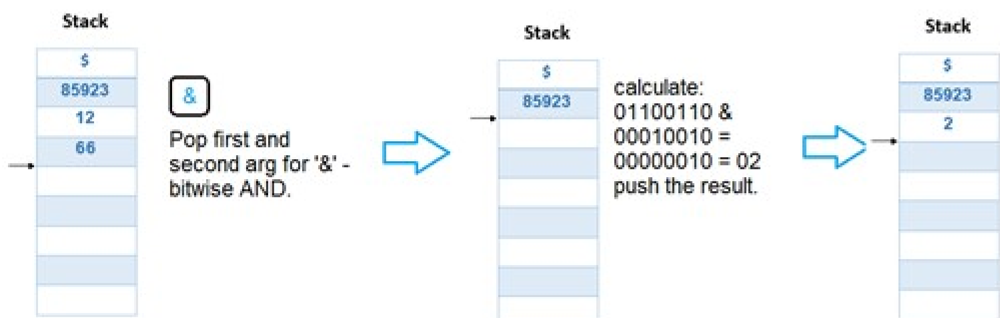
The (+) and (&) operators each get 2 operands, and provide one result. The "duplicate" operator takes one operand and provides two results - duplicates of its input operand. Note that all operands are implicit, i.e. they are popped from the stack, and not specified in the command line. The result(s) is pushed onto the stack. Pop-and-print takes one operand, and provides no result. It just prints the value of the operand to the standard output in **decimal**, as ASCII characters, of course.

Tip: You may want to use the **'daa'** instruction (see NASM manual) to help you do the addition in BCD. While you do not **have** to use this instruction, it may simplify your work considerably if you understand it…









Run example

An example of user input and program output appears below. Comments (which will not appear in input or output) are preceded by ";". The calculator prompt to the user is "calc: "

>>calc: 9 ; user inputs a number
>>calc: 1 ; user inputs another number
>>calc: d ; user enters "duplicate" operator
>>calc: p ; user enters pop-and-print-operator
>>1
>>calc: + ; user enters "addition" operator, 10 is in top of (and is the sole element in) stack right after
>>calc: d
>>calc: p
>>10

>>calc: 23 ; user enters another number 23
>>calc: +
>>calc: d
>>calc: p
>>33 ; the sum
>>calc: +
>>**Error: Insufficient Number of Arguments on Stack**
>>calc: 52 ; user inputs a number
>>calc: 93 ; user inputs a number
>>calc: & ; user enters Bitwise AND operator
>>calc: p
>>12 ; 12 = 00010010, the result of **01010010 & 10010011** "

>>calc: p
>>33
>>calc: q ; Quit calculator
11 ; Number of operations performed

Additional Requirements

Modularity is a requirement in this assignment. Thus, calculator functions, as well as input and output functions, must be programmed as procedures (subroutines). Additionally, printout of results should use the C library function printf(), and getting a line by using gets() or fgets(). Your code will be written **entirely in assembly language**. The "main" program (that you also need to write in assembly language) calls my_calc that is your primary procedure, and prints out the number of operations performed by my_calc. (Note that my_calc should count and return that number). If the user enters "q" at any time during the run of the program, your program should exit gracefully, by having my_calc returning the total number of operations performed, and using RET to return to the "main" code (which should print out that number before exiting).

In addition, you should implement a command line "-d" debug option. Your printout should look exactly as indicated above, except when the "-d" option is set, in which case you can and 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).

Desired Output:

- A successful calculation should present no additional output.
- Pop-and-print should print out the top member of the stack.
- If an action results in a stack overflow, your program must print (without the quotes): "**Error: Operand Stack Overflow**"
- If an action requires more arguments than currently available in the stack, your program should print: "**Error: Insufficient Number of Arguments on Stack**"
- if the input is not in the desire format(not an operation nor number), your program should print: "**Error: Illegal Input**"
- In any case, if an error occurs, your program must return the stack to its previous state (such as in a case when an action that requires 2 arguments fails because there is only one argument in the stack, etc).

Prototypes for C functions you can use

- char *fgets(char *str, int n, FILE *stream) // use fgets(buffer, BUFFERSIZE , stdin) to read from standard input
- int fprintf(FILE *stream, const char *format, arg list …) // use fprintf(stderr, …) to print to standard error (usually same as stdout)
- int printf(char *format, arg list …)
- void* malloc(size_t size) // size_t is unsigned int for our purpose
- void free(void *ptr)
- If you use those functions the beginning of your text section will be as follows (no _start label):

```
section .text
    align 16
    global main
    extern printf
    extern fprintf
    extern malloc
    extern free
    extern fgets
    extern stderr
    extern stdin
    extern stdout

main:
    … ; your code
```

- Declare a label "main:" and "global main" in your assembly program.
- Declare "extern printf, extern malloc" and "extern fgets" so you will be able to use those functions in the program. Note that you also need to declare as extern the appropriate FILE pointers, such as stdin, so that you can provide them to fgets( ), etc.

- 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.bin
```

Note: a C source file is not needed, but you are using C standard library so you need a global label "main" as shown above. Gcc will link external C library functions to your assembly program.

Assumptions and Minor requirements

- Illegal characters should be ignored.
- You may assume that the number entry format is correct.
- Each input line is no more than 80 characters in length, with the operator and/or number beginning as the first character of the line.
- Other errors can be either ignored or flagged, but <u>your program should never crash</u>.