$Id: asg4c-mydc-stackbignum.mm,v 1.32 2014-05-06 20:27:09-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Assignments/asg4c-mydc-stackbignum
URL: http://www2.ucsc.edu/courses/cmps012b-wm/:/Assignments/asg4c-mydc-stackbignum/

1. Overview

In this assignment you will implement a subset of the `dc`(1) arbitrary precision calculator. For specifications, read the man page for that utility, and experiment by running `dc` itself. You will implement six of its operators: `+`, `-`, `*`, `c`, `f`, `p`. Your program will be an executable image called `mydc`.

2. Modules in the program

The following modules are part of the program. For all but `main` a header (`.h`) file specifies the interface which is accompanied by an implementation (`.c`) file.

(a) Module `debug` contains useful debugging and tracing information.

(b) Module `stack` is a parameterized stack using an array implementation with array doubling to take care of a full stack.

(c) Module `bigint` is the important part of this project and handles multiprecision integer arithmetic using an array of characters.

(d) Module `token` reads long integers providing input to the rest of the program.

(e) Module `main` handles user interface, input and output.

3. Big integer implementation

Following is a more detailed discussion of how to implement the `bigint` module.

(a) Before attempting to implement `bigint`, perform each of the three operations on paper, reminding yourself how to perform the operations without a calculator.

(b) A `bigint` consists of an array of digits. Index 0 has the least significant digit, and the end of the array has the most significant digit. Each byte contains a single digit in the range 0 ... 9, inclusive. The capacity field specifies the dimension of the array, and the size field specifies the number of significant digits in the array, with leading zeros suppressed.

(c) Addition, if the signs are the same: call `do_add` to actually perform the addition and return a new `bigint`. Then set the sign to be the sign of one of the arguments.

(d) Addition, if the signs are different: call `do_sub` with the larger number as its left operand and the smaller number as the right operand. Then set the sign to that of the larger number.

(e) Subtraction: if the signs are different, call `do_add`, otherwise call `do_sub`.

(f) `Do_add` and `do_sub` are called from either the addition or subtraction function to do the array work. Note that it is marked `static` and is not called outside of the module.

(g) `Do_add` allocates a new `bigint` with space for a number of digits one larger than the largest operand. Then it loops across each array from index [0] to the end, adding and carrying as is done by hand:

```
digit = this->digits[index] + that->digits[index] + carry;
result->digits[index] = digit % 10;
carry = digit / 10;
```

There is a little extra trickiness at the high end of the shorter number.

(h) `Do_sub` allocates a new `bigint` whose size is the same as the left operand, and then performs the subtraction instead of addition:

```
digit = this->digits[index] - that->digits[index] - borrow + 10;
result->digits[index] = digit % 10;
borrow = 1 - digit / 10;
```

After computing the result, trim off high-order zeros.

(i) Multiplication proceeds by allocating a new `bigint` whose size is equal to the sum of the sizes of the other two operands. If $\mathbf{u}$ is a vector of size m and $\mathbf{v}$ is a vector of size n, then in $O(mn)$ speed, perform an outer loop over one argument and an inner loop over the other argument, adding the new partial products to the product $\mathbf{p}$ as you would by hand. The algorithm can be described mathematically as follows:

$$
\begin{array}{l}
\mathbf{p} \leftarrow \Phi \\
\textbf{for}\ \ i \in [0 \ldots m-1]: \\
\qquad c \leftarrow 0 \\
\qquad \textbf{for}\ \ j \in [0 \ldots n-1]: \\
\qquad\qquad d \leftarrow \mathbf{p}_{i+j} + \mathbf{u}_i \mathbf{v}_j + c \\
\qquad\qquad \mathbf{p}_{i+j} \leftarrow d \text{ remainder } 10 \\
\qquad\qquad c \leftarrow \lfloor d \div 10 \rfloor \\
\qquad \mathbf{p}_{i+n} \leftarrow c
\end{array}
$$

(j) The division and remainder algorithms are actually a single algorithm which produces both results, then discards the one not needed. This algorithm is complicated and not part of this assignment.

(k) Note that `malloc`(3) returns uninitialized storage, but `calloc`(3) sets its allocated storage to 0, so `new_bigint` calls `calloc`, not `malloc`, to allocate the underlying arrays. From the synopsis of `malloc`(3):

```
#include <stdlib.h>
void *calloc (size_t nmemb, size_t size);
void *malloc (size_t size);
void *realloc (void *ptr, size_t size);
void free (void *ptr);
```

4. Testing your program

Your program should write exactly the same output to both `stdout` and `stderr` as does `dc`(1), provided that inputs do not contain those facilities of `dc` that your

program is not expected to imitate. For example:

```
dc <test.in >test-dc.out 2>test-dc.err
mydc <test.in >test-mydc.out 2>test-mydc.err
diff test-dc.out test-mydc.out
diff test-dc.err test-mydc.err
```

Both of the **diff**(1) commands should produce no output for comparing **stdout**, and only a difference in the name of the ELF for **diff**ing **stderr**.

5. What to submit

Submit **Makefile**, **README**, and all C source and header files necessary for the grader to build your program with the command **make**. If you are doing pair programming, see the additional requirements.