CENG 320 Programming Assignment 1: Big Integer Library
By: Zackery Holloway

The objective of this programming assignment was to enhance the performance of the "bigint" library. Initially written in C, the library could achieve notable speed improvements through the conversion of specific functions into Assembly language. Although the original implementation catered to big integers with chunks of various bit lengths, only the segment relevant to 64 bits was required for this project.

After extracting the library and conducting the regression test, the baseline time recorded was 20.278718 seconds. Instructions for optimizing the bigint_cmp function were included at the end of the assignment. Utilizing the provided bigint_neg.S as a template, the optimization process commenced with the bigint_cmp function.

Upon completion, the Assembly version of the bigint_cmp function reduced the regression test duration to 11.080110 seconds, indicating a speedup factor of 1.82 for that particular function. The primary reason behind this acceleration likely stems from the C version's reliance on the bigint_sub function. In the C implementation, both large integers must undergo complete traversal to compute their difference before any comparisons can occur. In contrast, the Assembly version circumvents this process entirely and conducts chunk-by-chunk comparisons, starting from the most significant bit. Moreover, the Assembly version optimizes efficiency by only delving into the bigints as far as necessary. Additionally, the Assembly version initiates the comparison process by evaluating the signed bits and the number of chunks, potentially bypassing the chunk-by-chunk comparison altogether.

Here's how the Assembly version of the bigint_cmp function operates. Initially, the parameters are safeguarded by storing them in non-volatile registers. The dereferenced pointers of each bigint array, along with their respective sizes, are also retained in non-volatile registers. The first operation involves extracting the signed bit of each number. This is achieved by loading the most significant chunk from the array and using a logical shift right instruction to isolate the most significant bit. A value of 1 indicates a negative number, while 0 signifies a positive one. The function then branches based on the signed bit values. If one is positive and the other negative, the function sets w0 to the appropriate value (either 1 or -1) and returns. In cases where both numbers have the same signed value, the function proceeds to the next level of comparison, which involves examining their sizes

The size comparison principle relies on the notion that a number requiring more chunks to represent is greater in magnitude. Consequently, if both numbers are positive, the one with more chunks is deemed greater. However, in the case of both numbers being negative, the larger number in terms of chunks might actually have a greater negative magnitude, signifying that the smaller-sized number is indeed larger. This comparison step follows the branching from the signed bit evaluation and is executed swiftly in Assembly, facilitated by the early preservation of size into a nonvolatile register. Retrieving the size is straightforward due to the bigint struct storing the chunk count as an integer immediately after the array pointer. Thus, loading the size involves fetching the value of the first parameter offset by 8 into an integer register.

Only when both numbers exhibit the same sign and size does the function proceed to cycle through chunk-by-chunk comparison. Initially, a signed comparison of the most significant chunk is conducted, potentially yielding the appropriate 1 or -1. If equality is established, a loop begins, sequentially loading subsequent significant chunks and performing unsigned comparisons. The function halts and returns the appropriate value if any comparison indicates inequality. Should equality persist, the counter (initialized at size - 1) decrements, and the loop iterates. This process continues until either a disparity between chunks is detected or the counter reaches -1, denoting the conclusion of the number. The function only returns 0 if the counter reaches -1.

At any point during execution, if the function identifies disparity between the numbers, it sets the appropriate number to w0 and branches to "endloop". This label marks the conclusion of the function, after which nonvolatile registers are reloaded, and the function exits.

The programming assignment specified that the provided code folder should contain bigint_adc implemented in Assembly, serving as a template function. Despite its absence, its conspicuous omission highlighted it as the next target for optimization. The significance of this function became apparent upon a thorough examination of the bigint.c file. Also known as the "add with carry" function, bigint_adc serves as the cornerstone of all mathematical operations within the bigint library. It is utilized directly or indirectly in functions such as add, subtract, multiply, and negate.

As previously mentioned, bigint_adc operates as the "add with carry" function, accepting three parameters. The first two parameters represent bigint numbers to be added, while the third parameter is a standard integer denoting an initial carry value. Typically, this carry value is modified from 0 by negate and subtraction functions to accommodate negative numbers. The bigint_adc function facilitates the addition of all three numbers, forming a fundamental component of the mathematical operations within the library.

Upon completion, the Assembly version of the bigint_adc function reduced the regression test duration to 10.742551 seconds, resulting in a remarkable speedup factor of 1.92, nearly halving the runtime of the regression test. The likely reason for this acceleration is attributed to the difference in memory management between the C and Assembly versions. While the C version allocates and deallocates three new bigint structures, the Assembly version only allocates one and retains it as the returned value, eliminating the need for deallocation.

Here's how the Assembly version of the add with carry function operates: Initially, the carry value is handled, stored in a nonvolatile register for future use after the bigint_alloc function call. Subsequently, the sizes of both bigints are compared, then allocated.

The size difference between the two bigints is computed and replaces the larger size in the nonvolatile register. The previously prepared carry is moved into the PSTATE for utilization. The calculated difference in chunk sizes is then checked: if zero, indicating bigints of equal size, chunk-by-chunk addition is performed; otherwise, the function continues with signed bit addition to accommodate the larger bigint. Here, instead of loading the smaller number's chunks, the signed bit of the smaller chunk, arithmetically shifted right to encompass an entire chunk, is continuously added with carry. The previously calculated size difference between the chunk sizes serves as a counter, and loop_b continues until the counter reaches zero.

At this stage, the chunk-by-chunk addition process concludes, necessitating the handling of signs. Both signed bits are captured (potentially for the second time for the smaller bigint), added together with carry, and the result is stored in the most significant chunk of the new bigint.

The final task is to trim the bigint. Initially, one is subtracted from the size of the new bigint. If the most significant chunk solely comprises the signed bit of the next most significant chunk, it's redundant and can be trimmed using clean up. The most significant chunk of the new bigint is loaded and compared to the most significant bit of the next most significant chunk (shifted right to occupy an entire chunk). If the subtraction yields zero, indicating equality, the trim operation loops to recheck. If not, the function proceeds.

Before returning, one is added to the size of the new bigint to compensate for the arbitrary subtraction performed during trimming. As the x0 register has held the pointer to the new bigint since the call to allocate, reloading nonvolatile registers and returning complete the process.

Once more, the speedup of bigint_adc is attributed to the fact that the C version allocates and deallocates three new bigints, whereas the Assembly version handles only one allocation and no deallocation. Additionally, time is saved by the built-in trimming feature of the Assembly version.

This project underscores the efficiency benefits of Assembly. By rewriting just two functions, the speed of the program has nearly tripled. Although the converted functions were heavily utilized, time constraints prevented the rewriting of bigint_trim. With the deadline looming hours after the completion of this report, the necessary time for further optimizations is lacking.

Version	Time (seconds)	Speedup factor
C (Baseline)	20.278718	1.00
w/ Assembly bigint_cmp	11.080110	1.83
w/ Assembly bigint_adc	10.742551	1.88
w/ both Assembly functions	7.488271	2.61

CENG 320 Programming Assignment 1: Big Integer Library

By: Zackery Holloway