Adam Stammer

**Abstract:** Sometimes as programmers we are restricted in our instruction set, be it by the system, the application, time, memory, etc. Our task was to make a program to calculate the square of a given number without using our MUL instruction. We were given an algorithm that adds odd numbers together to find this square and output an error code of MIN\_INT in the case of overflow.

**Title:** N Squared (Program Three)

**Purpose:** To understand how to work within set confines and approach problems from multiple, sometimes unfamiliar, angles. To further familiarize ourselves with loops and branches. To increase our understanding of the varying costs, in both time and memory, of the instruction set.

**Inputs:** One positive 2’s complement integer (N as described in the pseudocode)

**Outputs:** One positive 2’s complement integer (N^2 or -32768 as error code)

**Pseudocode:**

START:

J = -1.

NSQ = 0.

TEMPN = N

IF TEMPN < 0

NSQ = -32768.

GOTO FINISH

LOOP:

TEMPN -= 1.

IF TEMPN < 0

GOTO FINISH

J += 2.

IF OVERFLOW

CRASH

NSQ += J

IF OVERFLOW

CRASH

GOTO LOOP

FINISH: END

**Discussion of Results and Lessons Learned:** For starters, I didn’t know this algorithm existed until now and I think it’s pretty cool! This assignment really made me think about the efficiencies of my loop designs. Depending on when you check the iterator and when you “prep” numbers to be used can drastically affect the amount of extra work done by a loop before it starts and after it’s done. On our modern machines, with our applications, it doesn’t have a noticeable effect, but in other applications it may. I’ve also found loops to be easier to debug when they are more efficient, in most cases, because of this.

My test cases started with negatives, especially the extremes to make sure they crashed with the proper error code. This wasn’t exhaustive but good enough for me. Then I checked the early working numbers, especially zero, to check for the expected output. Again, I got what I expected. My loop optimization had a significant impact on both TEMPN and on J, since this optimization decides when these numbers are modified.

Then I sought out the overflow fence. When does the answer become too big to store? Using known tools, I found that 181^2 is slightly under our 2’s comp. MAX\_INT, and that 182^2 is slightly over our 2’s comp. MAX\_INT. This fence was an obvious test choice, and ran as expected, but we’ve run into the same problem as I had on the last assignment: I used preexisting solutions to find the limits of a known problem. I could’ve directly found the limit for our algorithm, or used the age old guess and check method, but it was much easier to just use a calculator. This fence did behave as expected. From there I moved to expected overflow cases and high extremes to check for expected error codes and possible unanticipated behavior.

Attached is the generated list file of my program and a copy of the log file running through the majority of my test cases showing both input and output.