**Group 13**

**Members: Agustin, Gedrite H.**

**Bravante, Ralph Christian S.**

**Chua, Kyle Matthew C.**

**COMORGA S11**

**MP1 DOCUMENTATION:**

Our group has been tasked with the implementation of the Taylor Series of the trigonometric sin function given below:

To implement the said function, our group has constructed an algorithm which allows our program to correctly calculate each iteration of the Taylor series. To make it easier to formulate our algorithm, we used the divide and conquer method, and we arrived at the conclusion that we should separate the numerator and the denominator.

The flow of our program is as follows:

* Getting the input from the user (In radians)
* Loading values inputted by the user and the required initialization values for the stack
* Start of the loop for the final result and the value for each iteration
* Getting the results for the numerator by getting the square of the radians and alternating sign of each iteration by multiplying the number by negative one
* Since the stack has a copy of the previous iteration’s factorial, we only need to add the current iteration’s factorial
* Dividing the numerator and the denominator and storing the result
* Printing of the result for the current iteration
* Adding the result to the total

To implement the algorithm, the group utilized float stack registers to store the required variables specifically the current value, total, factorial value, denominator, numerator, input radian, -1, and 1. As stated before, the algorithm of the fraction was separated into two parts, numerator and denominator, and patterns for each part were identified separately. For the numerator, the -1 raised to n is just simply changing the sign of the term for each iteration. The group also noticed that for the x raised to 2n+1, each iteration has an interval of power of 2 which denotes that for every iteration, the numerator is just multiplied by the input radian twice, thus coming up with this algorithm for the numerator shown below:

where variable numerator stores the value of the numerator of the previous iteration. The denominator on the other hand also has an interval of 2 for each iteration. For this part, the group will first get the value of the denominator before computing for its factorial. To achieve efficiency, the group will not compute the factorial of the denominator for each iteration because it is repetitive, so the group decided to store the previous computed factorial value and will be used in the next iteration. For example, iteration two has the factorial of 3 (containing 3\*2\*1), so instead of computing for the factorial of 5 in iteration 3 (also containing 3\*2\*1) the group will just multiply the two new digits to the previous factorial (3! \* 4 \* 5) thus also achieving factorial of 5 with much efficient algorithm. The algorithm for the denominator is shown below:

where variable factorial contains the computed factorial value of the previous iteration. To combine the numerator and denominator, the group just simply divide the two values and store it to ST0 (iteration value) which will then be added to ST1 (total value). The unique part of our algorithm is that we do not pop the values from the stack registers to the memory using FST to reduce confusion, instead, the group exchanges the needed stack register to ST0 (because all of the arithmetic functions for stack must have ST0 as a destination or source) and after performing the arithmetic, the stack register is exchanged back to its original place. The illustration of the stack register and its variable equivalent (See Figure 1).

|  |  |
| --- | --- |
| **ST0** | Iteration Value |
| **ST1** | Total Value |
| **ST2** | Factorial |
| **ST3** | Denominator |
| **ST4** | Numerator |
| **ST5** | Input radian |
| **ST6** | -1 |
| **ST7** | 1 |

Figure 1. Stack floating point registers representation