Lab-3: Taylor Series using RISC-V F extension

Pathri Vidya Praveen
CS24BTECH11047

September 21, 2025

1 Introduction

This lab assignment involves the implementation of five common mathematical and scientific functions using Taylor series expansions in the **RISC-V Assembly** with the **Floating-Point (F) extension**. The functions implemented are the exponential function (e^x), sine ($\sin(x)$), cosine ($\cos(x)$), natural logarithm ($\ln(x)$), and reciprocal ($1/x$). The program is designed to take input from memory for the function code, a floating-point value as argument to fucntion , and the number of terms to be used in the taylor series expansion. A key design decision was to compute each term iteratively within the main loop, avoiding separate, less efficient functions for power and factorial calculations. This approach, while less modular, directly computes each term from the previous one, leading to more efficient code. The implementation also handles the unique characteristics of each series, such as alternating signs and domain restrictions, and includes error handling for invalid inputs.

---

2 Design Approach

The program first takes memory input at 0x10000 input N, the number of results to compute. For each of the N inputs, the user provides a function code (0 for e^x, 1 for $\sin(x)$, 2 for $\cos(x)$, 3 for $\ln(x)$, and 4 for $1/x$), a single-precision floating-point value, and the number of terms for the series. The main function then calls the appropriate Taylor series function based on the provided code. And then it stores out put at 0x10000200

A core principle of this implementation is the direct calculation of each Taylor series term from the previous one. Instead of computing powers and factorials from scratch for each term, the code uses loop counters and simple arithmetic to derive the next term in the series from the current one. This strategy significantly reduces computational overhead. For instance, in the e^x series, the k-th term, $\frac{x^k}{k!}$, can be found by multiplying the $(k-1)$-th term, $\frac{x^{k-1}}{(k-1)!}$, by $\frac{x}{k}$.

2.1 Function Implementations

The following formulas describe the Taylor series expansions used for each function:

1. **Exponential Function (e^x)**: The Taylor series for e^x is given by:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

   This series is straightforward to implement as all terms are positive. We simply add each new term to the running total. The next term is calculated by multiplying the previous term by $\frac{x}{n}$, where n is the current loop iteration.

2. **Sine Function ($\sin(x)$)**: The Taylor series for $\sin(x)$ is given by:

$$\sin(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

   To handle the alternating signs, a conditional branching approach is used. The program checks the parity of the loop counter using a bitwise AND operation. If the counter is even, the term is added to the sum; if it is odd, the term is subtracted. This avoids the use of a separate floating-point register for sign tracking.

3. **Cosine Function ($\cos(x)$)**: The Taylor series for $\cos(x)$ is given by:

$$\cos(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

   Similar to the sine function, the terms of the cosine series also alternate in sign. The sign tracking mechanism is handled in the same way as for $\sin(x)$, by checking the loop counter's parity and branching accordingly.

4. **Natural Logarithm ($\ln(x)$)**: The Taylor series for $\ln(x)$ around $x = 1$ is given by:

$$\ln(x) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{(x-1)^n}{n} = (x-1) - \frac{(x-1)^2}{2} + \frac{(x-1)^3}{3} - \dots$$

   The series is valid for $0 < x \leq 2$. The code handles invalid inputs ($x \leq 0$ or $x > 2$) by returning NaN. A similar conditional branching approach is used to handle the alternating signs.

5. **Reciprocal Function ($1/x$)**: The Taylor series for $1/x$ is given by the geometric series expansion for $1/(1-(1-x))$, valid for $|1-x| < 1$, which simplifies to $0 < x < 2$. The expansion is:

$$\frac{1}{x} = \sum_{n=0}^{\infty} (1-x)^n = 1 + (1-x) + (1-x)^2 + (1-x)^3 + \dots$$

The program checks for the valid domain of x, returning NaN for values outside the range.

---

3 Issues Encountered

During development, several challenges were faced:

- **RARS Simulator**: Initial setup and familiarization with the RARS simulator environment during development and shift to RARS from RIPES.
- **Register Management**: A significant issue was the misuse of registers, led to incorrect results and NaN outputs. Debugging these issues required careful tracking of register usage and ensuring values were not overwritten . This resulted in hours of debugging.
- **Floating-Point Conversion**: Conversion between floating-point and integer representations was a reason of many errors. Initially, `fmv` instructions were used, but it was discovered that these instructions copy the bit pattern rather than performing a true type conversion.
- **Code Modularity**: A key decision was to avoid separate functions for power and factorial, leading to a more direct and efficient approach. However, this choice resulted in code that is less modular and potentially less readable. Future improvements could focus on optimizing this approach while maintaining clarity.

---

4 Verification Approach

To ensure the correctness of the implementation, the following verification approach is used:

- **Online Converters**: For low-level verification of floating-point representations, an online tool (e.g., Floating Point to Hex Converter) was used to check the hexadecimal representation of floating-point values, confirming correct data handling.
- **Python Scripting**: A Python script was written to compute Taylor series values for a given function, input, and number of terms. The output from the RISC-V program was then compared against the values from this script, providing a check for accuracy.
- **Edge Case Testing**: Specific edge cases were tested to validate the program's error handling. This included providing inputs outside the valid domains for the $\ln(x)$ and $1/x$ functions to confirm that the program correctly returns NaN (Not a Number).