Computer Structure:

# Assignment 1
# Introduction to Assembly Language

Course 2024/2025


**Héctor Molina Garde.**
100522253.
100522253@alumnos.uc3m.es.
Group 88.
Computer Science Engineering Degree

**Nicolás Maire Bravo**.
100522100.
100522100@alumnos.uc3m.es.
Group 88.
Computer Science Engineering Degree

# Table of content

# Exercise 1

---

## Program behaviour and design

The program's structure is divided into two key functions: **Compute_Integral** for handling the integral calculation and **f** for calculating the quadratic function at a given point.

A loop iterates over “N” steps, updating the running total of the function evaluations, and finally multiplies this sum by “h” to approximate the integral.

The program separates the logic of evaluating the quadratic function from the integration process, making it easy to manage and reuse. The stack is used effectively to save and restore registers, ensuring safe execution without overwriting useful values during function calls, following RISC-V passing conventions for register and memory management. The use of floating-point operations ensures the precision necessary for integration, and the inclusion of an auxiliary function **f** that calculates the quadratic expression facilitates the integration process by allowing the integral function to focus only on the numerical method of approximation with the summary.

The design ensures that integer arguments (representing the polynomial coefficients and integration limits) are converted into floating-point numbers for accurate computation, providing precision when integrating.

The passing convention has been correctly used, using s-type registers, as we are working with non-terminal functions, and the correct use and management of a-type registers starting from a0 and fa0 to introduce just the needed arguments when calling the functions. Furthermore, all the relevant values are efficiently stored in s-type registers to not be overwritten when executing the functions. Additionally, we have been concerned with the registers reuse, trying to reuse as much as possible but keeping a clear and understandable code. To enhance the understanding of the code, we have followed a rational order of the register’s subindexes trying to keep arguments grouped, and auxiliary and final operations using the last subindexes, so that when reading the code, you start from the smallest to the greatest.

Overall, the program's design is clear as it is a commented code, efficient, follows modular programming, and aligns with the exercise's goals of implementing both the numerical integration and the evaluation of the quadratic function while following proper register and stack implementations.

## Pseudocode

```
def f(p: int, q: int, r: int, x: float) -> float:
    #Evaluate the polynomial p*x^2 + q*x + r
    f: float = 0
    a: float = pow(x, 2)  #x^2
    a = a * float(p)
    f = f + a
    b: float = x * float(q)
    f = f + b
    f = f + float(r)
    return f

def Compute_Integral(a: int, b: int, p: int, q: int, r: int, N: int) -> float:
    #Compute the integral of the polynomial from a to b using N subdivisions
    h: float = (b - a) / N
    n: int = 0
    sum: float = 0

    while n < N:
        x: float = n * h
        x = x + a
        func = f(p, q, r, x)
        sum = sum + func  #Accumulate the function values
        n = n + 1

    sum = sum * h  #Multiply the accumulated sum by h to get the total integral
    return sum
```

# Battery tests

| Data to enter | Description of the test | Result expected (approximation) | Result obtained (stored in fa0) |
|---|---|---|---|
| $\int_2^4 (2x^2 + 3x + 1)\ dx$ | A normal defined integral (base case) | 57.33 | f10 \| fa0 `57.30333...` |
| $\int_8^2 (x^2 + 4x + 2)\ dx$ | Integral in which the lower bound is greater than the upper bound | -300 | f10 \| fa0 `-300.252...` |
| $\int_1^3 (2x^2 - 4x - 1)\ dx$ | Integral dealing with negative sign | −0.6667 (-2/3) | f10 \| fa0 `-0.67466...` |
| $\int_2^2 (2x^2 - 4x - 1)\ dx$ | Integral with upper bound equal to lower bound | 0 | f10 \| fa0 `0` |
| $\int_1^4 7\ dx$ | Integral of a constant function | 21 | f10 \| fa0 `21` |
| $\int_{-2}^2 (x^2 + 1)\ dx$ | Integral with negative bound | 9.333 (28/3) | f10 \| fa0 `9.333341...` |
| $\int_5^{999999999} (x^2 + 3x + 2)\ dx$ | Integral with a bound close to infinite | $3.333 \times 10^{26}$ $\infty$ | f10 \| fa0<br>Hex. `000000006B89A823`<br>Binary `00000000000000000000000(`<br>IEEE 754 (32 bits) `3.3283368098473775e+26`<br>IEEE 754 (64 bits) `8.91385129e-315` |

# Exercise 2

## Program behaviour and design

The main part of the program is the **Integral_Matrix** function, which uses the **Compute_Integral** function to fill up an MxM matrix with calculated integral values based on certain conditions. The matrix gets filled according to the following setup:

a. matrix[a][b] = $\int_b^a f(x)dx$ when b > a
b. matrix[a][b] = 0 when b ≤ a

The **Integral_Matrix** function takes the address of a single-precision floating-point values matrix and integers M, p, q, and r. Here, M is the matrix size, while p, q, and r are the coefficients for the polynomial $f(x) = px^2 + qx + r$, evaluated over intervals for each matrix element. The interval count N is set to 100 for consistent integration steps.

The function is designed to keep things simple and organized by using **Compute_Integral** to handle the integration work separately. This makes **Integral_Matrix** focused on filling in the matrix, while **Compute_Integral** takes care of the calculations. This split makes the code easier to understand and reuse in other parts of the program.

The function starts by adjusting the stack pointer and pushing some important registers onto the stack. This step is needed to keep the register values safe and avoid overwriting them by mistake while the function is running. s-type registers and the return address (ra) are saved, creating a setup to handle matrix calculations and the function calls of **Compute_Integral**.

The matrix's offset calculation is essential for finding the right memory spot for each element. The offset for matrix[a][b] is calculated as (a * M + b) * 4, which uses the base address adjusted by rows and columns. The function then goes through the matrix using two loops (one for rows and one for columns): if a < b, the function sets up a, b, p, q, r, and N as arguments and then calls **Compute_Integral** using the 'jal' instruction. It stores the result in matrix[a][b] and then jumps past the 'else' block to avoid running unnecessary code. If a >= b, the element in the matrix is just set to zero.

After filling in the matrix, the function restores all the saved registers from the stack and resets the stack pointer before finishing with a 'jr ra' instruction to return to the calling function. This careful handling of registers helps keep the function clean, efficient, and organized, following RISC-V conventions. Floating-point operations are used for the integration since they give the precision needed, especially with single-precision values, and integer arguments are converted when necessary. Grouping values by their roles in the code makes it easier to understand and follow, making the function efficient, readable, and aligned with conventions for stack and register management.

# Pseudocode

**Pseudocode in python language**

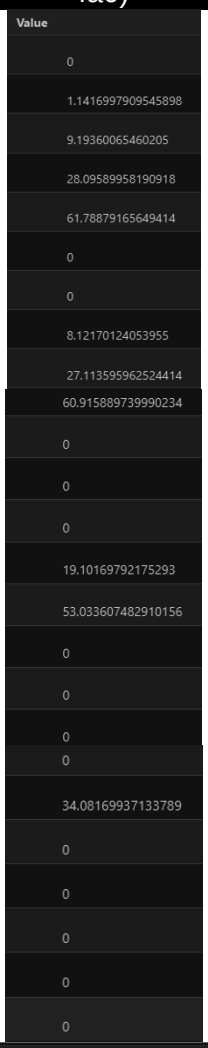
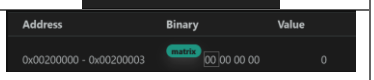
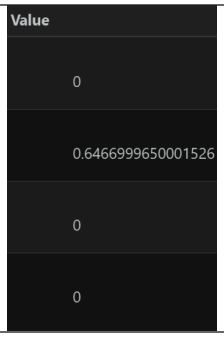
```
def Integral_Matrix(*matrix: pointer, M: int, p: int, q: int, r: int):
    """
    Python
    Function to fill a matrix with integral values based on a condition.
    Inputs:
    - matrix: pointer to the starting address of a single-precision floating-point matrix
    - M: dimension of the matrix
    - p, q, r: coefficients for the function f(x) = p*x^2 + q*x + r
    """
    N = 100  # Number of intervals for computing the integral

    # Loop through rows
    for a in range(M):
        # Loop through columns
        for b in range(M):
            if b > a:
                # Compute the integral and store in matrix[a][b]
                matrix[a][b] = Compute_Integral(a, b, p, q, r, N)
            else:
                # Set the matrix value to 0 if b <= a
                matrix[a][b] = 0
```

---

**Pseudocode thought to be implemented in assembly**

```
def Integral_Matrix(*matrix: int, M: int, p: int, q: int, r: int):
    """
    Psudocode(python with C pointers and not using 'for' loops)
    Function to fill a matrix with integral values based on a condition.
    Inputs:
    - matrix: pointer to the starting address of a single-precision floating-point matrix
    - M: dimension of the matrix
    - p, q, r: coefficients for the function f(x) = p*x^2 + q*x + r
    """
    matrix: float = [[]] #bidimensional array size MxM
    N = 100
    a = 0
    while a < M:
        b = 0
        while b < M:
            &matrix_pointer = *(matrix + a * M + b)
            if b > a:
                matrix_pointer = Compute_Integral(a, b, p, q, r, N)
            else:
                matrix_pointer = 0
            b += 1
        a += 1
```

# Battery tests

| Data to enter | Description of the test | Result expected (approximation) | Result obtained (stored in fa0) |
|---|---|---|---|
| M = 5 (5x5 matrix)<br>$f(x) = 2x^2 + 3x - 1$ | M>1 (base case) | 0.00 1.67 9.33 28.50 62.67<br>0.00 0.00 8.17 27.33 61.50<br>0.00 0.00 0.00 19.17 53.33<br>0.00 0.00 0.00 0.00 34.17<br>0.00 0.00 0.00 0.00 0.00 |  |
| M = 1 (1x1 matrix)<br>$f(x) = 2x^2 + 3x - 1$ | M = 1 (most simple matrix) | 0.00 |  |
| M = 2 (2x2 matrix)<br>$f(x) = 2x^2 + 2x - 1$ | M = 2 (first matrix size with result != 0) | 0.00 0.6667<br>0.00 0.00 |  |

# Conclusions

This project has been useful to practise the contents seen in class, specifically handling with calling functions and managing stocks. Furthermore, we have been able to improve our ability to work with Creator tool and practice finding errors by debugging the code, which have also made us more familiar with the assembly language and the possible errors that can be encountered and where they can be founded.

The fact that the work has been done in pairs have let us improve our way to work in groups, not only reading, correcting and analysing my teammate code, but also to find the errors and try to look for the solutions as a team, which have been really enriching.

In terms of programming, generally, we have been able to program the base of the code relatively fast. The problem that has delayed our work has been working with stocks and identifying minimal errors about some register's names or some minimal inconsistencies that made the code don't work. For example, when working during a made version of the code, we used to change some registers and add others, but we forgot to update the pop and push or we updated wrong, then passing conversions error were always made.

As an estimation, the time spent in this project has been about 18 hours one and over 12 hours the other partner, then an average of 15 hours each, which make a total of 30 hours. The principal time spent has been in proof and error tests, finding new errors each time and changing in some cases many registers names or moving others, which increases the time as each name must be changed each appearance in the code.