

## **Portada**

Proyecto:

Calculadora Vectorial en Visual Studio

Autores:

Braunny Madrigal Barrantes (C24436)

Marcelo Picado Leiva (C15950)

Henoc Rojas Carrillo (C26764)

Kendall Villafuerte Beita (C28387)

Curso:

CI-0118 Lenguaje Ensamblador

Institución:

Universidad de Costa Rica

Escuela:

Escuela de Ciencias de la Computación e Informática

Profesor:

Dr. Carlos Vargas

Ciclo:

I Ciclo del 2024

## **Contenido**

### **1. Descripción del problema**

Se programa la simulación de una “Calculadora Vectorial” que opera con números reales, utilizando programación híbrida (mezcla de lenguaje Visual C++ y ensamblador), corriendo en modo protegido, moderna arquitectura Intel X64. El desarrollo de la aplicación se realiza en el Ambiente de Desarrollo Integrado (IDE) “Visual Studio Community” versión actual, utilizando MASM como ensamblador.

### **2. Alcances y limitaciones del programa**

## **Alcances:**

El programa "Calculadora Vectorial" permite realizar las siguientes operaciones:

1. Suma de dos vectores de números flotantes.
2. Multiplicación de un escalar por un vector de números flotantes.

Las operaciones se realizan utilizando programación híbrida, combinando C++ para la interfaz de usuario y ensamblador únicamente para los procedimientos de AVX que maximizan el procesamiento en paralelo en arquitectura X64. El código fuente se desarrolla en Visual Studio Community con soporte para MASM, aprovechando las capacidades de registros XMM y ZMM según sea necesario. Lo importante de este proyecto es ver la comunicación de código de alto nivel (C++) con código ensamblador específico de alguna arquitectura concreta (AVX 512, en este caso).

## **Limitaciones:**

El formato estricto de entrada requiere que los elementos de los vectores se ingresen separados por comas, sin espacios, y la máxima cantidad de elementos que un vector puede almacenar debe ser un número múltiplo de 8.

Otra limitación, es que en el caso de la suma de vectores se decide programar dicha operación de tal forma que solo se haga con vectores de igual tamaño. Esta decisión se considera lógica ya que si se quisiera sumar un vector de mayor tamaño con uno de menor tamaño; en esta implementación bastará con llenar el de menor tamaño con valores 0, hasta que tenga el mismo número de elementos que aquel de menor tamaño. En resumen, esto no es ningún limitante para el usuario que desee usar la calculadora.

La calculadora puede realizar una sola operación a la vez y la precisión decimal con la que imprime los resultados es variable. Ya que no se especificó que tanta precisión decimal se desea

que compute la calculadora, se decidió trabajar con métodos estándar de C++. En resumen la precisión decimal de los resultados vectoriales estará condicionada por los procedimientos “std::stod(std::string);” y por “std::cout << double;” de la biblioteca estándar de C++.

Las operaciones de AVX en ensamblador están limitadas a procesar vectores de 8 elementos a la vez, lo anterior es cierto ya que se decidió trabajar con los registros ZMM y operar allí con precisión decimal doble. Lo cual implica que los vectores deben tener una longitud múltiplo de 8 para ser procesados eficazmente desde C++.

### 3. Algoritmos utilizados

En este programa hay muy pocos procedimientos que tengan un mínimo de complejidad algorítmica a nivel implementación. Pero los que sí lo tuvieron, se decidió hacer un pseudocódigo en estilo python para desarrollarlos antes de convertirlos a lenguaje ensamblador.

Algoritmo que suma dos vectores de números flotantes utilizando instrucciones AVX.

```
def AVXSumTwoVectors(myVector1, myVector2):  
    # Simulando los registros ZMM como listas de números flotantes  
    ZMM0 = myVector1[:] # Copia del primer vector  
    ZMM1 = myVector2[:] # Copia del segundo vector  
  
    # Suma utilizando instrucciones AVX  
    ZMM2 = [ZMM0[i] + ZMM1[i] for i in range(len(ZMM0))]  
  
    # Actualiza el primer vector con el resultado de la suma  
    myVector1[:] = ZMM2  
    return # retorna con el myVector1 ya modificado
```

Algoritmo que multiplica un vector de números flotantes por un escalar de tipo flotante utilizando instrucciones AVX.

```
def AVXMulVecByScalar(myVector, myNumber):  
    # Simulando los registros ZMM y XMM como listas y variables de números  
    # flotantes  
    ZMM0 = myVector[:] # Copia del vector
```

```

XMM1 = myNumber      # Escalar

# Hace un broadcast del escalar a un vector del mismo tamaño que ZMM0
ZMM1 = [XMM1] * len(ZMM0)

# Multiplica el vector por el escalar
ZMM2 = [ZMM0[i] * ZMM1[i] for i in range(len(ZMM0))]

# Actualiza el vector con el resultado de la multiplicación
myVector[:] = ZMM2
return # retorna con el myVector ya modificado

```

Algoritmo que muestra cómo se crea la interfaz que maneja la operación de sumar creada en MASM usando las instrucciones AVX de INTEL.

```

def sum_interface():
    try:
        vector_size = int(input("Ingresa la cantidad de elementos de ambos
vectores: "))
        if vector_size % 8 != 0:
            raise ValueError("La cantidad de elementos ingresada no es
multiplo de 8.")

        vector1 = list(map(float, input("Ingresa el primer vector:
").split(',')))
        if len(vector1) != vector_size:
            raise ValueError("El número de elementos no llena al vector.")

        vector2 = list(map(float, input("Ingresa el segundo vector:
").split(',')))
        if len(vector2) != vector_size:
            raise ValueError("El número de elementos no llena al vector.")

        num_segments = vector_size // 8
        for seg in range(num_segments): # suma de 8 en elems del vector,
respetar la convención
            AVXSumTwoVectors(vector1[seg * 8 : (seg + 1) * 8], vector2[seg
* 8 : (seg + 1) * 8])

```

```

        print("Resultado:", ", ".join(map(str, vector1)))
    except ValueError as e:
        print(f"Error: {e}")
    return # retorna

```

Algoritmo que muestra cómo se crea la interfaz que maneja la operación de multiplicar creada en MASM usando las instrucciones AVX de INTEL.

```

def mul_interface():
    try:
        vector_size = int(input("Ingresa la cantidad de elementos del
único vector: "))
        if vector_size % 8 != 0:
            raise ValueError("La cantidad de elementos ingresada no es
multiplo de 8.")

        vector1 = list(map(float, input("Ingresa el vector:
").split(',')))
        if len(vector1) != vector_size:
            raise ValueError("El número de elementos no llena al vector.")

        my_double = float(input("Ingresa el escalar: "))

        num_segments = vector_size // 8
        for seg in range(num_segments): # de 8 en 8, limitación de AVX
            AVXMulVecByScalar(vector1[seg * 8 : (seg + 1) * 8], my_double)

        print("Resultado:", ", ".join(map(str, vector1)))
    except ValueError as e:
        print(f"Error: {e}")
    return

```

#### 4. Estructura del programa

##### Descripción de constantes y de variables

El método en cuestión es “void tests()” de la clase Interface. A continuación, se enumeran y describen las variables y constantes utilizadas en este método:

myOption (int)

1. Inicializado a -1.
2. Variable para almacenar la opción seleccionada por el usuario en el menú.
3. Controla el bucle while, terminando cuando el valor es 0.

El método “int getOption()” de la clase Interface obtiene una opción del usuario desde la entrada estándar, la convierte a un número entero y la retorna. A continuación, se enumeran y describen las variables utilizadas en este método:

myOption (int)

1. Inicializado a -1.
2. Variable para almacenar la opción convertida a entero desde la entrada del usuario.
3. Se retorna al final del método.

myInput (std::string)

1. Inicializado a una cadena vacía "".
2. Variable para almacenar la entrada del usuario como una cadena de texto.
3. Utilizada para leer la opción ingresada por el usuario.

El método “void sumInterface()” de la clase Interface se encarga de sumar dos vectores de números flotantes, solicitando al usuario el tamaño de los vectores y sus elementos. A continuación, se enumeran y describen las variables utilizadas en este método:

vectorSize (int)

1. Inicializado a 0.
2. Almacena el tamaño de los vectores, especificado por el usuario.
3. Debe ser múltiplo de 8 para usar instrucciones AVX.

userInput (std::string)

1. Inicializado a una cadena vacía "".
2. Almacena la entrada del usuario como una cadena de texto

vector1 (double)\*

1. Inicializado a nullptr.
2. Puntero al primer vector de números flotantes.

vector2 (double)\*

1. Inicializado a nullptr.
2. Puntero al segundo vector de números flotantes.

memoryAsked (bool)

1. Inicializado a false.
2. Indica si se ha solicitado memoria para los vectores.

ss (std::stringstream)

1. Inicializado a un nuevo std::stringstream.
2. Utilizado para procesar la entrada del usuario.

value (std::string)

1. Inicializado a una cadena vacía "".
2. Utilizado para almacenar los valores individuales de la cadena de entrada al descomponerla.

i (int)

1. Inicializado a 0.
2. Utilizado como índice para recorrer y asignar valores a los vectores.

numSegments (int)

1. Inicializado a 0.
2. Almacena el número de segmentos de 8 elementos en el vector, utilizado para las operaciones AVX.

El método “void mulInterface()” de la clase Interface se encarga de multiplicar un vector de números flotantes por un escalar, solicitando al usuario el tamaño del vector, los elementos del vector, y el escalar. A continuación se describen las variables utilizadas en este método:

vectorSize (int)

1. Inicializado a 0.
2. Almacena el tamaño del vector, especificado por el usuario.
3. Debe ser múltiplo de 8 para usar instrucciones AVX.

userInput (std::string)

1. Inicializado a una cadena vacía "".
2. Almacena la entrada del usuario como una cadena de texto.

vector1 (double)\*

1. Inicializado a nullptr.
2. Puntero al vector de números flotantes.

myDouble (double)

1. Inicializado a 0.
2. Almacena el valor del escalar ingresado por el usuario.

memoryAsked (bool)

1. Inicializado a false.
2. Indica si se ha solicitado memoria para el vector.

ss (std::stringstream)

1. Inicializado a un nuevo std::stringstream.
2. Utilizado para procesar la entrada del usuario.

value (std::string)

1. Inicializado a una cadena vacía "".
2. Utilizado para almacenar los valores individuales de la cadena de entrada al descomponerla.

i (int)

1. Inicializado a 0.
2. Utilizado como índice para recorrer y asignar valores al vector.

numSegments (int)

1. Inicializado a 0.
2. Almacena el número de segmentos de 8 elementos en el vector, utilizado para las operaciones AVX.



El método main es el punto de entrada del programa. A continuación, se enumeran y describen las variables utilizadas en este método:

interface (Interface)

1. Se declara una instancia de la clase Interface.
2. Utilizada para acceder a los métodos welcome y tests de la clase Interface.

El método “bool AVXFoundationDetection()” es una rutina en ensamblador que verifica la compatibilidad del procesador con las extensiones vectoriales avanzadas (AVX). A continuación, se enumeran y describen las variables utilizadas en este método:

- No se utilizó ninguna variable con este método.

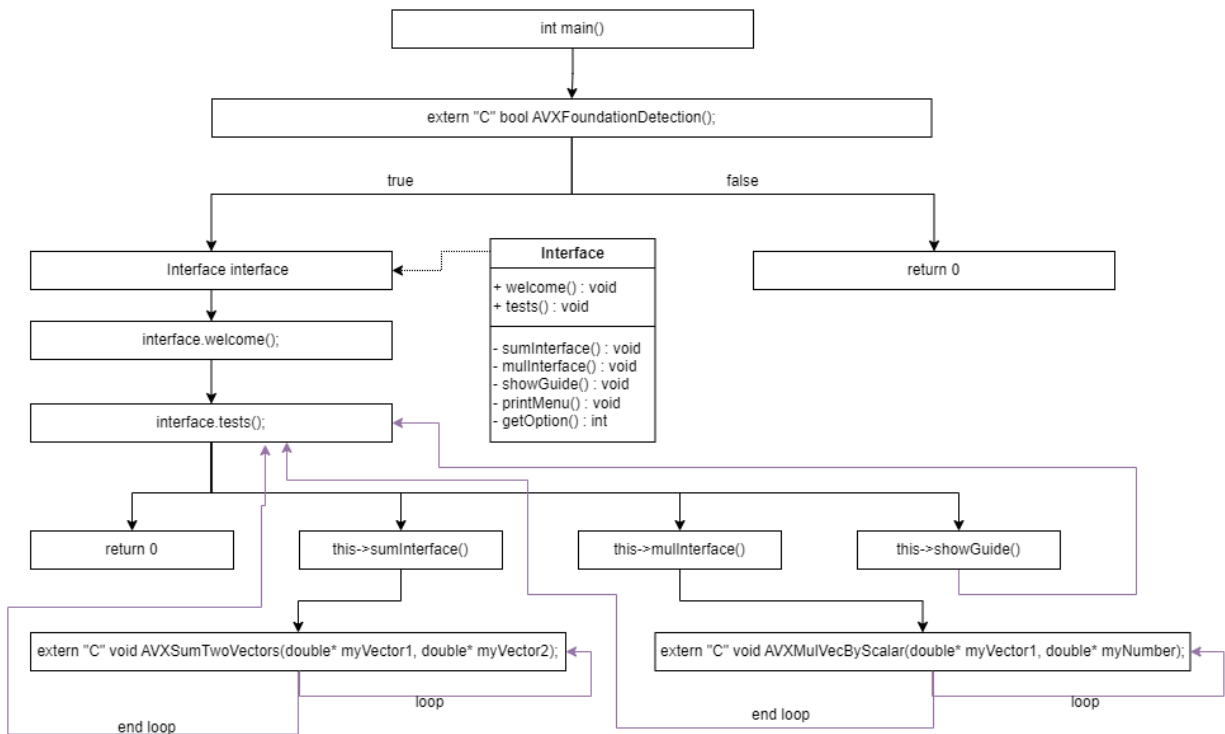
El método “void AVXSumTwoVectors(double\* myVector1, double\* myVector2)” es una rutina en ensamblador que suma dos vectores de números flotantes utilizando instrucciones AVX. A continuación, se enumeran y describen las variables utilizadas en este método:

- RCX: Puntero al primer vector (myVector1). Este parámetro se pasa como registro para acceder a los datos del primer vector.
- RDX: Puntero al segundo vector (myVector2). Este parámetro se pasa como registro para acceder a los datos del segundo vector.

El método “void AVXMulVecByScalar(double\* myVector1, double\* myNumber)” es una rutina en ensamblador que multiplica un vector de números flotantes por un escalar utilizando instrucciones AVX. A continuación, se enumeran y describen las variables utilizadas en este método:

- RCX: Puntero al vector (myVector1). Este parámetro se pasa como registro para acceder a los datos del vector.
- RDX: Puntero al escalar (myNumber). Este parámetro se pasa como registro para acceder al valor del escalar.

## Procedimientos



## 5. Detalles importantes de la implementación

**Optimización para AVX:** El programa aprovecha las instrucciones AVX para realizar operaciones intensivas de manera eficiente en arquitecturas Intel X64. Esto se logra mediante el uso de registros ZMM para procesar múltiples elementos de vectores de números flotantes simultáneamente, mejorando significativamente el rendimiento computacional.

**Validación de entrada estricta:** La entrada de datos para los vectores debe seguir un formato estricto sin espacios entre los elementos y separados por comas. Esta validación asegura que las operaciones de suma y multiplicación se realicen correctamente, manteniendo la integridad de los datos y cumpliendo con los requisitos de AVX para operaciones eficientes.

**Manejo de precisión decimal:** Las operaciones en el programa están diseñadas para manejar la precisión decimal utilizando tipos de datos estándar de C++. Esto garantiza resultados precisos y

coherentes según los estándares de redondeo y manejo de errores definidos por la biblioteca estándar.

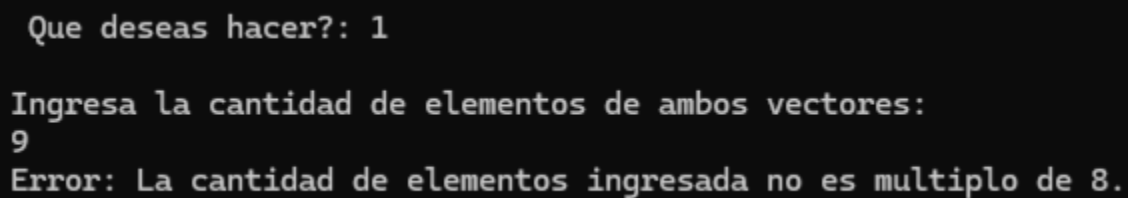
Arquitectura modular: El programa se estructura en módulos separados para la interfaz de usuario en C++ y las rutinas críticas en ensamblador AVX. Esta modularidad facilita el mantenimiento del código y la optimización individual de cada componente, asegurando un desarrollo escalable y robusto del software.

Compatibilidad y detección de AVX: Se incluye una rutina de detección de capacidades AVX durante la inicialización del programa. Esto garantiza que el software solo intente utilizar instrucciones AVX en procesadores compatibles, mejorando la portabilidad y la experiencia del usuario en diferentes configuraciones de hardware.

Estos detalles destacan los aspectos clave de diseño e implementación que hacen que la "Calculadora Vectorial" sea eficiente, precisa y adecuada para su despliegue en entornos modernos de computación intensiva.

## 6. Datos de prueba utilizados

Tamaño de vector incorrecto. Se ingresa como tamaño de vector un número que no es múltiplo de 8.



```
Que deseas hacer?: 1
Ingresa la cantidad de elementos de ambos vectores:
9
Error: La cantidad de elementos ingresada no es multiplo de 8.
```

Vector incorrecto. Se ingresa en alguna posición del vector un elemento que no es un número.

```

Que deseas hacer?: 1

Ingresa la cantidad de elementos de ambos vectores:
8
Ingresa el primer vector:
12,45.4,0,0,0,f,0,0
Error: Entrada no valida. No es un numero valido.

```

Computar la suma de vectores que contiene únicamente números positivos.

```

Que deseas hacer?: 1

Ingresa la cantidad de elementos de ambos vectores:
16
Ingresa el primer vector:
12.345,2.3,5.78,8.9,10.11,15.6,3.14,6.78,9.0,11.22,13.45,4.56,7.89,16.7,18.9,20.1
Ingresa el segundo vector:
1.1,2.2,3.3,4.4,5.5,6.6,7.7,8.8,9.9,10.0,11.1,12.2,13.3,14.4,15.5,16.6
Resultado:
13.445,4.5,9.08,13.3,15.61,22.2,10.84,15.58,18.9,21.22,24.55,16.76,21.19,31.1,34.4,36.7

```

Computar la suma de vectores que contienen tanto números positivos como negativos.

```

Que deseas hacer?: 1

Ingresa la cantidad de elementos de ambos vectores:
32
Ingresa el primer vector:
1.5,-2.7,3.9,-4.1,5.3,-6.5,7.7,-8.9,9.1,-10.3,11.5,-12.7,13.9,-14.1,15.3,-16.5,17.7,-18.9,19.1,-20.3,21.5,-22.7,23.9,-24.1,25.3,-26.5,27.7,-28.9,29.1,-30.3,31.5,-32.7
Ingresa el segundo vector:
1.0,-2.0,3.5,-4.8,5.2,6.9,-7.1,8.4,9.6,-10.2,11.3,-12.5,13.7,14.0,-15.8,16.4,-17.2,18.9,19.1,-20.6,21.3,-22.0,23.5,24.7,-25.9,26.1,27.4,-28.6,29.8,30.0,-31.2,32.5
Resultado:
2.5,-4.7,7.4,-8.9,10.5,0.4,0.6,-0.5,18.7,-20.5,22.8,-25.2,27.6,-0.1,-0.5,-0.1,0.5,0,38.2,-40.9,42.8,-44.7,47.4,0.6,-0.6,-0.4,55.1,-57.5,58.9,-0.3,0.3,-0.2

```

Computar la multiplicación por escalar con un vector que contiene únicamente números positivos.

```

Que deseas hacer?: 2

Ingresa la cantidad de elementos del unico vector:
8
Ingresa el vector:
1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0
Ingresa el escalar:
3.3
Resultado:
3.3,6.6,9.9,13.2,16.5,19.8,23.1,26.4

```

Computar la multiplicación por escalar con un vector que contiene números positivos, pero también negativos.

```
Que deseas hacer?: 2
Ingresa la cantidad de elementos del unico vector:
32
Ingresa el vector:
1.0,-2.0,3.5,-4.8,5.2,6.9,-7.1,8.4,9.6,-10.2,11.3,-12.5,13.7,14.0,-15.8,16.4,-17.2,18.9,19.1,-20.6,21.3,-22.0,23.5,24.7,-25.9,26.1,27.4,-28.6,29.8,30.0,-31.2,32.5
Ingresa el escalar:
2.7
Resultado:
2.7,-5.4,9.45,-12.96,14.04,18.63,-19.17,22.68,25.92,-27.54,30.51,-33.75,36.99,37.8,-42.66,44.28,-46.44,51.03,51.57,-55.62,57.51,-59.4,63.45,66.69,-69.93,70.47,73.98,-77.22,80.46,81,-84.24,87.75
```

## 7. Análisis de resultados (indicar grado de funcionamiento)

El análisis de resultados de la "Calculadora Vectorial" se centra en evaluar la precisión, eficiencia y funcionalidad de las operaciones implementadas. A continuación se detalla el grado de funcionamiento observado durante las pruebas y evaluaciones del programa:

**Precisión Decimal y Manejo de Errores:** Las operaciones de suma y multiplicación por escalar mantienen la precisión decimal adecuada, utilizando las funciones estándar de conversión y salida de C++. Esto asegura que los resultados sean consistentes y precisos, cumpliendo con los estándares de redondeo y manejo de errores.

**Eficiencia de Operaciones AVX:** Las rutinas en ensamblador optimizadas permiten procesar múltiples elementos simultáneamente, aprovechando al máximo la capacidad de los registros ZMM en arquitectura X64.

**Validación de Entrada y Gestión de Errores:** El programa valida estrictamente la entrada de datos, asegurando que los vectores ingresados cumplan con los requisitos de tamaño múltiplo de 8 y formato correcto (sin espacios y separados por comas). Esta validación previene errores durante las operaciones y mantiene la integridad de los datos manipulados.

**Compatibilidad y Detección de AVX:** Asegura una experiencia de usuario consistente en diferentes configuraciones de hardware.

**Pruebas y Validación de Casos de Uso:** Se han realizado pruebas exhaustivas utilizando diversos conjuntos de datos, incluyendo casos con vectores de diferentes tamaños y contenidos mixtos de

números positivos y negativos. El programa ha demostrado manejar correctamente estas situaciones, proporcionando resultados esperados y coherentes.

En resumen, la "Calculadora Vectorial" cumple satisfactoriamente con los objetivos de precisión, eficiencia y manejo de errores, ofreciendo una herramienta robusta y efectiva para operaciones vectoriales en entornos de computación avanzada.

## **8. Breve Guía del Usuario.**

- Descargar “Visual Studio Community”. El siguiente enlace lo lleva directo al repositorio oficial del programa mencionado. <https://visualstudio.microsoft.com/>
- Instalar “Visual Studio Community”, para ello seguir las recomendaciones que se dan en la página a la cual redirige el enlace anterior.
- Abrir la herramienta previamente descargada. Dentro de “Visual Studio” seleccionar la opción “Open a project or solution”. Ahora, dirijase a la carpeta llamada “Tarea2Ensambla” y escoja el archivo “.sln”.
- Una vez abierto el proyecto, identifica la barra llamada “Solution Explorer”. En dicha barra de click derecho al icono que representa su proyecto, seguido de, darle click a la opción “Build Dependencies” y “Build Customizations”. En la ventana emergente marcar con un ‘check’ la opción llamada “masm”.
- Presione “Ctrl + F5” para compilar y ejecutar el programa.
- Siga las instrucciones que la interfaz de consola programada en C++ le indique.