PRÁCTICA 1
INTRODUCCIÓN AL LENGUAJE ENSAMBLADOR

CURSO 2023/2024

22 DE OCTUBRE DE 2023

AUTORES:
PABLO AMOR MOLINA / 100495855 / 10495855@ALUMNOS.UC3M.ES
MANUEL ROLDÁN MATEA 100500450 /100500450@ALUMNOS.UC3M.ES
GRUPO 81 INGENIERÍA INFORMÁTICA
Estructura de computadores

Índice de contenidos

1. Ejercicio 1

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

1.1 Funciones seno y coseno

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

El desarrollo de las subrutinas del seno y el coseno ha sido la parte más laboriosa de la práctica, puesto que para su correcto funcionamiento nos hemos visto obligados a apoyarnos en otras tres subrutinas: *simplificador*, *exponente* y *factorial.*

Antes de explicar el funcionamiento de la subrutina principal, creemos conveniente el introducir dichas subrutinas auxiliares de las que hacemos uso en distintas ocasiones.

Para la función seno usamos principalmente registros s, mientras que en para las subrutinas, al ser terminales, usamos registros t, a excepción de en alguna variable de la subrutina *simplificador*, que utilizamos s por si en algún momento hubiese podido ser necesario otra subrutina dentro de esta.

Simplificador:

Esta subrutina la desarrollamos previamente de ser informados de que bastaba con que las funciones *sen* y *cos* calculasen valores entre -π/2 y π/2, por ello, aunque ya no fuese estrictamente necesaria, por el trabajo que nos supuso hemos decidido conservarla, permitiendo que nuestras funciones *sen* y *cos* devuelvan resultados precisos para cualquier valor.

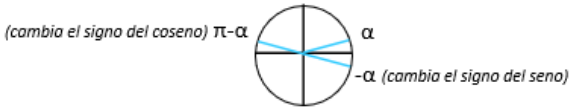
El objetivo de esta subrutina es el de reducir cualquier número a otro en el rango (-π/2, π/2) en el caso del seno y (-π, π) en el caso del coseno, con seno y coseno equivalentes, para así evitar desbordamientos y poder operar con cualquier valor. Recibe como parámetros el número a reducir y 1 o 0, indicando si opera con un seno o un coseno, respectivamente, y devuelve el número reducido.

```
def simplificador(x: float, op: int) -> float:

    pi = 3.141592
    pimedios = pi / 2
    i = 0

    if abs(x) > pimedios:

        if x < 0:
            while abs(x) > pimedios:

                x = x + pi
                i += 1

        elif x > 0:
            while abs(x) > pimedios:

                x = x - pi
                i += 1

        if i%2 == 1:  # Ajuste del signo

            if op == 1: # Seno
                x = -x

            elif op == 0: # Coseno

                if x <= 0:
                    x = pi + x

                else:
                    x = pi - x

    return x
```

Esto lo consigue mediante un bucle que suma, si es negativo o resta, si es positivo al parámetro el valor de π con cada iteración, hasta que este tenga un valor absoluto inferior a π/2. Posteriormente, evalúa si el número de iteraciones ha sido impar (es decir, que se han dado “x vueltas y media” a la circunferencia) y de ser así, hay que hacer cálculos en base a si estamos operando un seno o un coseno.

En el caso del seno, simplemente se cambia el signo del valor, (equivalente a sumar 180º)

En el caso del coseno, se resta el valor actual a 2π si es positivo y se le suma si es negativo. Esto sirve para hacer un “espejo” respecto a la vertical, moviéndonos así solo 1 cuadrante. (Ver dibujo). Para valores de x entre |π/2| y |π| esto resulta en deshacer el bucle previo, quedándonos con el resultado inicial, puesto que no se habría podido simplificar más, sin embargo, hemos decidido no implementar esta excepción para no saturar más el código.



Factorial:

El objetivo de esta subrutina es el de calcular al margen de la función principal el factorial de un número. Hemos decidido separarlo en otra subrutina para así prescindir de repetir código a lo largo de la práctica, puesto que la función factorial se usa en numerosas ocasiones. Recibe como parámetros el número del que calcular su factorial y devuelve dicho factorial.

Para llevar su función a cabo, la subrutina realiza un bucle cuyo límite de iteraciones es el parámetro introducido, multiplicando desde 1 hasta el mismo todos los números enteros entre ambos, guardando el resultado en otro registro. Esta subrutina solo opera con números enteros, puesto que el parámetro introducido siempre será entero (debido a que es el número de la iteración del sumatorio).

```
def factorial(x:int) -> int:
    result = 1
    i = 1
    while i <= x:
        result = result * i
        i += 1

    return result
```

Exponente:

El objetivo de esta subrutina es calcular el resultado de elevar a cierto exponente cierta base. Hemos decidido implementarlo por simplificar el código, de manera análoga al factorial. Esta subrutina recibe como parámetros la base (coma flotante) y el exponente (entero) y devuelve el resultado de realizar la potencia (coma flotante).

El funcionamiento de esta se basa también en un bucle cuyo límite de iteraciones es el exponente introducido. En cada una de ellas, se multiplica el resultado actual (comenzando en 1) por la base introducida como parámetro.

```
def exponente(base: float, exp: int) -> float:

    result = 1.0
    i = 0
    while i < exp:

        result = result * base
        i += 1

    return result
```

Una vez explicados estas tres subrutinas de apoyo, podemos pasar al desarrollo de las subrutinas seno y coseno propiamente dichas.

Lo primero que realiza cada una es llamar a la función *simplificador* ya con el parámetro en fa0, para manejar ya un número acotado en nuestro rango. Hemos decidido realizar 10 iteraciones en el sumatorio, para obtener mayor precisión en el resultado. Dentro del bucle de dicho sumatorio, se realizan las siguientes operaciones:

1. Se calcula el exponente de la x y el parámetro del sumatorio, que será 2n + 1 en el caso del seno y 2n en el caso del coseno, siendo n la iteración que se está ejecutando.
2. Se llama con el resultado anterior como parámetro a la subrutina *factorial*, con la que obtendremos el denominador de la operación.
3. Se llama a la subrutina exponente, con el número simplificado y el resultado del paso 1 como parámetros, que devuelve el numerador sin signo.
4. Se calcula el signo del numerador en base a la paridad de la iteración, mediante la operación "resto de 2" (si es par, se deja como está, mientras que si es impar se le agrega un signo negativo). Esta operación hace referencia al $(-1)^n$ de la fórmula de Taylor.
5. Convertimos el denominador a coma flotante para realizar la división y sumamos el valor obtenido de esta al contenido del registro en el que guardamos el resultado del sumatorio.
6. Sumamos una iteración y volvemos al inicio del bucle.
7. Una vez terminado el bucle, se mueve el resultado del sumatorio al registro fa0, terminando la función.

```
def sin(x: float) -> float:

    resultado = float(0)

    x = simplificador(x, 1)

    n = 0
    while n < 7:

        exp = 2*n
        exp = exp + 1

        denominador = factorial(exp)

        numerador = exponente(x, exp)

        if n%2 == 1:
            numerador = numerador*(-1)

        resultadotemp = numerador / denominador

        resultado += resultadotemp
        n += 1

    return resultado
```

```
def cos(x: float) -> float:

    resultado = float(0)

    x = simplificador(x, 0)

    n = 0
    while n < 7:

        exp = 2*n

        denominador = factorial(exp)

        numerador = exponente(x, exp)

        if n%2 == 1:
            numerador = numerador*(-1)

        resultadotemp = numerador / denominador

        resultado += resultadotemp
        n += 1

    return resultado
```

1.2 Tangente

Una vez realizadas las dos funciones anteriores de sin y cos, la dificultad de la función tg se reduca drásticamente. La fórmula usada para realizar la función es sin/cos que una vez aproximados los valores del sin y cos es una simple operación. Además se ha implementado que la función devuelva como resultado infinito en el caso de que el cos esté muy cerca de 0.

El proceso realizado para la función *tg* ha sido el siguiente

1. Guardamos el valor inicial recibido para no perderlo.
2. Llamamos la función sin para que realice el seno del valor y guardamos el resultado.
3. Llamamos a la función coseno para que realice el coseno y guardamos el resultado.
4. Comprobamos si el valor absoluto del coseno es < 0,0001; de ser así, devuelve infinito.
5. Una vez guardados los dos resultados se realiza una división con los valores guardados.
6. Se manda el resultado de la división al parámetro de salida en fa0.

```
def tg(x: float):

    seno = sin(x)
    coseno = cos(x)

    if abs(coseno) < 0.0001:
        result = math.inf

    else:
        result = seno / coseno

    return result
```

1.3 Número E $e = \sum_{n=0}^{\infty} \frac{1}{n!}$

El desarrollo de Taylor del número E es mucho más simple que el del seno y el del coseno. En esta función solo se hará uso de la subrutina *factorial*, que servirá para calcular el denominador en cada iteración.

El número de iteraciones del sumatorio que hemos escogido en este caso es 7, pues con este valor la precisión del resultado se ajustaba a lo establecido por el enunciado en nuestro código. Dentro del bucle del sumatorio se realiza las siguientes operaciones:

1. Se llama a la función *factorial* con el número de iteración (n) como parámetro.
2. Se transforma el resultado devuelto por *factorial* a coma flotante para realizar la división.
3. Se hace la división 1 / n! y se suma el resultado de esta al registro que contiene el resultado general del sumatorio.
4. Una vez finalizado el bucle, se devuelve el resultado del sumatorio en fa0.

```
def E() -> float:

    resultado = float(0)

    n = 0
    while n < 7:

        denominador = factorial(n)
        resultadotemp = 1 / denominador

        resultado += resultadotemp
        n += 1

    return resultado
```

2. Ejercicio 2

2.1 Función SinMatrix

La implementación de esta función ha sido más trivial que las anteriores ya que no hay que implementar nuevas subrutinas auxiliares, simplemente usamos la función sin del ejercicio 1 como subrutina de la función principal SinMatrix.

Sin embargo, hay algunos factores a tener en cuenta, principalmente los parámetros de la función sin tener .data ni main han sido las únicas dificultades, pero para solventarlo hemos creado los nuestros auxiliares para ayudarnos a visualizar la trazabilidad de la función (ver anexo).

La función hace uso de un doble bucle para iterar la matriz, tratando el primer bucle con las filas(N) y el segundo con las columnas(M). Para poder iterar las matrices a la vez en ensamblador se necesitan guardar las direcciones de memoria del primer valor de cada una. Una vez guardada esa dirección sumamos 4 a cada dirección y accedemos al siguiente valor de cada matriz. Por último destacar que si las dos matrices no fueran de la misma dimensión no se podrían realizar la transformación a la matriz de los senos.

Una vez sabiendo como se iteran las matrices sólo queda realizar la operación:

Esta función realiza: **B[i,j] = sin(A[i,j]);**

Para realizar la operación, usamos el segundo bucle para iterar cada elemento de cada matriz.

1. Se usa la instrucción para cargar valores (lw) de memoria, se carga el valor de la primera matriz(A).
2. Se llama a la función sin y se pasa por parámetro el valor cargador anteriormente
3. Se carga el valor(sw) en la dirección de la segunda matriz(B).
4. Se suman 4 a las direcciones de memoria de cada una de las matrices para que coincidan las posiciones de las matrices.
5. Se incrementa j hasta que el bucle acabe.
6. Se incrementa i cuando j se acabe entonces se ha recorrido una columna, para recorrer las demás es importante volver a establecer j a 0 cada vez que se incrementa i.

```
def SinMatrix(matrixA:list, matrixB:list, N:int, M:int):

    i = 0
    while i < N:
        j = 0
        while j < M:
            matrixB[i][j] = sin(matrixB[i][j])
            j += 1
        i += 1
```

3. Batería de pruebas

Ejercicio 1

Datos a introducir:	Descripción de la prueba:	Resultado esperado:	Resultado obtenido:
0	Evaluamos los resultados con sen(x) = 0	Sin: 0,0 Cos: 1,0 Tan: 0,0	Sin: 0,0 Cos: 1,0 Tan: 0,0
π/2 (1,57079)	Evaluamos los resultados con cos(x) = 0	Sin: 1,0 Cos: 0,0 Tan: ∞	Sin: 0,99994367 Cos: 0,00001028 Tan: Infinity.0
1	Número positivo sin iteraciones en *simplificador*	Sin: 0,84147098 Cos: 0,540302306 Tan: 1,55740772	Sin: 0,84147095 Cos: 0,54030227 Tan: 1,55740773
-1	Número negativo sin iteraciones en *simplificador*	Sin: -0,84147098 Cos: 0,540302306 Tan: -1,55740772	Sin: -0,84147095 Cos: 0,54030227 Tan: -1,55740773
2,5	Número positivo con número de iteraciones impares en *simplificador*	Sin: 0,59847214 Cos: -0,801143616 Tan: -0,747022297	Sin: 0,59847223 Cos: -0.80113947 Tan: -0.74702626
-2,5	Número negativo con número de iteraciones impares en *simplificador*	Sin: -0,59847214 Cos: -0,801143616 Tan: 0,747022297	Sin: -0,59847223 Cos: -0.80113947 Tan: 0.74702626
5	Número positivo con número de iteraciones pares en *simplificador*	Sin: -0,95892427 Cos: 0,283662185 Tan: -3,380515006	Sin: 0,95892429 Cos: 0,28366202 Tan: -3,380501700
-5	Número negativo con número de iteraciones pares en *simplificador*	Sin: 0,95892427 Cos: 0,283662185 Tan: 3,380515006	Sin: 0,95892429 Cos: 0,28366202 Tan: 3,380501700

Ejercicio 2

Datos a introducir:	Descripción de la prueba:	Resultado esperado:	Resultado obtenido:
3 2,2 5 1 7,2 4 0 0 5,5 Matriz A ↑ N = 3; M = 3; B todo 0s	Matrices simétricas	0,14112001 0,8084964 -0,9589243 0,84147098 0,79366786 -0,7568025 0 0 -0,7055403	0,1411201 0,8084965 -0,9589243 0,841471 0,7936675 -0,7568025 0 0 -0,7055404
10 0 -3,3 33 7 2 Matriz A ↑ N = 2; M = 3; B todo 0s	Matrices asimétricas	-0,5440211 0 0,15774569 0,9999119 0,6569866 0,90929743	-0,54402071 0 0,15774556 0,99991196 0,65698641 0,90929746
5 0 3,3 1,5 Matriz A ↑ 1 1 1 1 Matriz B ↑ N = 2; M = 2	Matriz B con valores distintos de 0	-0,95892429 0 -0,15774556 0,99749511	-0,9589243 0 -0,1577457 0,99749499

4. Conclusiones, problemas encontrados y tiempo estimado

Esta práctica nos ha resultado muy beneficiosa a la hora de introducirnos en el mundo del ensamblador. Empezamos a desarrollar el trabajo en profundidad la semana de antes del primer miniexamen y lo cierto es que nos permitió realizar con mucha soltura las partes de ensamblador de este.

Los principales problemas los hemos encontrado a la hora de introducir a ensamblador los números en coma flotante, porque nos faltaba mucha práctica y al principio apenas sabíamos manejar los enteros. Tras esto, apareció el problema del desbordamiento, por el que tuvimos la idea de implementar el simplificador, con el que estuvimos discutiendo mucho por como desarrollarlo y trabajando mucho el cómo funcionaba el sumar o restar valores en la circunferencia goniométrica, tanto para el seno, como para el coseno, que gracias a las pruebas pudimos recordar que no se comportaban igual respecto a los cuadrantes y los signos.

También, como muchos otros compañeros, estos últimos días hemos tenido muchas dudas acerca del formato de entrega de los códigos. No nos terminaba de quedar claro por la forma en la que quedaba redactado tanto en distintos correos como en las instrucciones, lo que sumado a nuestra inexperiencia con la herramienta nos ha planteado muchas dudas y confusión.

Nos quedamos con que gracias a esta práctica creemos que hemos podido comprender con bastante profundidad el cómo funciona el set de instrucciones de RISC-V, así como su convenio establecido, que seguro que nos servirá de gran utilidad en el futuro

Sección	**Tiempo invertido**
Pseudocódigo	2 horas 30 minutos
Funciones Sin y Cos	2 horas
Número E	1 hora
Subrutinas exponente y factorial	1hora 30 minutos
Subrutina simplificador	3 horas
Función SinMatrix	1 hora 30 minutos
Práctica 1	**Total: 11 horas 30 minutos**