Universidad de Santiago de Chile

Arquitectura de Computadores (13309-0-A-1)

Laboratorio 2 Acercándose al Hardware

Docentes: Leonel Medina
Manuel Villalobos Cid
Ayudante: Maximiliano Orellana

Estudiante: Nícolas Farfán Cheneaux

Diciembre 2021

Contenido

Introducción

Problema

Los problemas por desarrollar son los siguientes:

- Encontrar el máximo de 2 números ingresados por la consola.
- Algoritmo de Euclides (recursivo) para hallar el MCD de 2 números dados.
- Algoritmo de la multiplicación y división utilizando solo operadores de suma y resta (La división debe retornar una división decimal si es que esta es impropia).
- Aproximación de función seno, logaritmo natural con expansiones de Taylor, usando las subrutinas creadas en el punto anterior.

Objetivos

El objetivo final, consiste en implementar las instrucciones que se piden en MIPS, haciendo uso correcto del stack al usar la recursividad y los registros de retorno de las subrutinas, así como empelar buenas prácticas al momento de limpiar los registros usados en la subrutina.

- Aprender a usar correctamente las subrutinas.
- Hacer correcto uso del Stack al usar las subrutinas.
- Construir programas funcionales que cumplan con lo que se pide.
- Utilizar el punto flotante en MIPS y ver el funcionamiento del coprocesador.

Marco Teórico

Para resolver los presentes problemas se requiere entender los conceptos de Stack, recursividad en MIPS.

Conceptos Importantes:

- Stack: Es una estructura de datos, del tipo *LIFO* que significa *Last In First Out*, el cual indica que el ultimo dato ingresado a esta será el primero que esté disponible para salir.

- Recursividad: El procedimiento que ocurre cuando una función o subrutina realiza un proceso invocándose a si misma, modificando los valores de entrada en cada llamado a sí misma, añadiendo este estado al Stack de memoria, dejando los llamados casos pendientes, para que este proceso termine de ejecutarse debe existir un caso base, el cual resuelve el último caso pendiente (TOP stack), y al estar unidos recursivamente, resuelve todo el stack.
  Tipos de Recursividad:
  Existes 3 tipos usados actualmente:
  - Natural: Se produce cuando se debe utilizar la memoria para dejar estados pendientes en el stack, como el caso del factorial recursiva (Ver anexo) se le llama recursión natural.

  - De cola: A diferencia de la anterior, esta no deja estados pendientes por resolver en el stack, ya que los parámetros se operan directamente en el llamado de la función o subrutina, dejando el caso base dentro de estos, es decir que no espera a resolver otro resultado, como es el caso de la factorial con recursión natural, sino que lo resuelve en cada llamado. (Ver Anexo) para ejemplo.

  - Arborea: Esta recursión tiene la misma estructura que una recursión natural, salvo la excepción que como su nombre indica, crece de manera exponencial en cada llamado recursivo, un ejemplo es la sucesión de Fibonacci (Ver Anexo).

- Subrutina: Es una función o proceso que ejecuta dentro del conjunto de instrucciones.
- Serie de Taylor: Es una serie infinita de potencias las cuales representan una aproximación de funciones notables, para términos de este laboratorio usaremos 2 variaciones de esta.
  (Ver Anexo)

Explicación de la solución

Problema 1: Calcula el máximo entre 2 números ingresados por consola.

En este problema se usó solamente 1 subrutina aparte de 'main' llamada "greater" la cual realiza un salto condicional dependiendo si a > b salta a la subrutina que imprime "a" como mayor.
Sino imprime "b"

Instrucciones relevantes usadas:

Slt $t1, $t2, $t3: Set Less Than, coloca 1 en $t1 si $t2 es menor que $t3, caso contrario coloca 0
bnez: Branch Equal Zero: La cual salta a mayor $s1 si $s1 es mayor que $s2

Problema 2

Pseudocódigo algoritmo Euclídes
valor absoluto |a| y |b|

```
function mcd(a, b):
        if b == 0:
                retorna a
        else:
                return mcd(b, a%b)
```

Nota: El algoritmo de Euclides solo funciona para números positivos (Ver Anexo),

Para llegar a "adaptar" este algoritmo recursivo a MIPS, se consideró exactamente las mismas condiciones (caso base y recursivo). Además es necesario crear por cuenta propia un "stack" para almacenar los llamados recursivos o estados pendientes, asignado al registro **$sp** (Stack Pointer).
Por lo tanto al final de la subrutina se debe limpiar el stack por buenas prácticas.

Caso Base:

```
beqz $a1,  mcdListo
```

Caso Recursivo:

```
div $a0, $a1
mfhi $t3
```

Donde mfhi se usa para sacar el módulo

Problema 3a

Para la multiplicación entera de a * b se hace la suma de "a", "b" veces, quedando un bucle de forma:

```
function multiplicacion (a, b):
        a = aux
        if (b == 0)
                retornar b

        mientras(b != 0)
                a = a + aux
                b = b - 1
        retornar a
```

Función cambio de signo:

```
signSwitch:
        # Sign Cases
        bltz        $a0, casemM
        bltz        $a1, caseMm
        j           continue                # Case ( +)( + )
        casemM:
            # Case ( - ) ( + )
            bltz        $a1, casemm
            add         $t1, $a0, $a0
            sub         $a0, $a0, $t1
            li          $t0, 1              # Control Register
            j           continue

        casemm:
            # Case ( - ) ( - )
            add         $t1, $a0, $a0
            sub         $a0, $a0, $t1
            add         $t1, $a1, $a1
            sub         $a1, $a1, $t1
            j           continue

    caseMm:
        # Case ( + ) and ( - )
        add         $t1, $a1, $a1
        sub         $a1, $a1, $t1
        li          $t0, 1              # Control Register

    continue:
        jr          $ra
```

Para cambiar el signo se tiene un controlador el cual dice si existe un negativo en alguno de los 2 números guardando 1 en $t0 para hacerle valor absoluto al resultado de sea cual sea la operación que se esté realizando.

Se puede resumir en 2 casos:

1) Si existe algún negativo entre ellos, pero solo uno lo es, se guarda un "controlador" o verificador para indicar que eso está pasando y poder cambiarle el signo al resultado final.
2) Si ambos son negativos o positivos se procede a no hacer ningún cambio de signo.

Problema 3b

Para la división de a / b, se hace el uso de un contador, el cual registra el número de veces que el número "b" se puede restar a "a", obteniendo un tipo de recursión de cola.

```
function divisionParteEntera (a, b):
        if (b == 0)
                retorna null
        else:
                if ( a - b<= 1)
                        retorna cont
                else:
                        retorna division ((a – b), b, (1 + cont))
```

Proceso en MIPS:

```
.data
    f1:     .float 0.1              # For "multiply" units without mul
    f2:     .float 0.01             # For "multiply" hundreds without mul
    one:    .float 1                # Auxiliar to load 1 in coproc1
```

1) Se carga en los registros $f1 y $f2 los números 0.1, 0.01 para multiplicar los residuos de la división Parte Entera, almacenados en $a1.
2) Se ejecuta la división entera, la cual retorna un residuo y un cociente
   Casos:
   - Si el residuo es distinto de 0 se debe hacer la parte decimal.
   - Si el residuo es 0 pero a < b entonces se debe hacer la parte decimal.

3) Si la división no es entera se usa el coprocesador y se multiplica el residuo (unidades por * 0.10). Y (centenas por * 0.01) dentro del coprocesador.
4) Finalmente se cambian los signos correspondientes.

Problema 4

Este ejercicio pide calcular la serie de Taylor con las subrutinas usadas anteriormente, una cosa a considerar es que los registros solo pueden almacenar una factorial de 13! Como máximo por lo tanto esto nos limita al momento de calcular la serie de el sin(x).
Así que implementaremos 2 versiones de este problema: una con la división y multiplicación entera el cual solo llegará a

orden 5. Acto seguido implementaremos nuevas funciones de multiplicación división, potencia, factorial para usar el coprocesador y poder almacenar números grandes.

División Recursiva Entera

El objetivo de esta sección es explicar el uso de el stack en el algoritmo de la división, como se puede ver se asigna un espacio en el stack para los 2 parámetros de entrada “a” y “b” los cuales se “actualizarán” en cada llamado a la función, y una vez el caso base es cumplido se termina la función, nótese que en cada llamado se cargan los valores del stack para continuar con el procedimiento.

```
integerDiv:
        beqz    $a1, exitFunctionD      # If $a1 = 0 the algorithm can't execute
        beqz    $a0, exitFunctionD      # If $a0 = 0 return 0
        recursiveDiv:
            subu    $sp, $sp, 12        # Stack space assignement
            sw      $ra, ($sp)
            sw      $s0, 4($sp)
            sw      $s1,  8($sp)

            sub     $t1, $a0,$a1        # Base Case
            bltz    $t1,divDone         #  If (a + b<= 1)

            move    $s1, $a1
            move    $s0, $a0

            sub     $a0, $a0, $a1       # a = (a - b)
            jal     recursiveDiv
            addi    $v1,$v1 ,1          # (v1 += 1)

        divDone:
```

División Flotante sin dividir ni multiplicar.

Para implementar esta función es necesario hacer el algoritmo de la división el cual agrega una coma en caso la división no es entera, pero realmente que es agregar una coma?

Por ejemplo si se quiere dividir 50/7, podemos calcular la parte entera sin mayores complicaciones con nuestra implementación de división entera, pero la parte decimal es otro aspecto, en este caso tenemos una parte decimal de 7, la cual puede ser almacenada en un registro para más adelante.

Seguidamente tenemos el resto de 50%7=1, entonces multiplicamos este por 10 y lo dividimos nuevamente por 7, obteniendo un cociente de 1 y un resto de 3, como obtuvimos un cociente lo “agregamos” a la parte decimal pero cómo? Es eso en realidad? Lo que se está produciendo es una multiplicación por la unidad de las “unidades” valga la redundancia, ya que ese 1 se convierte en 0,10, si lo multiplicamos por 0,10, y si lo sumamos tenemos 7,10.

Tenemos una división decimal sin dividir ni una sola vez. Para calcular las centenas se repite este mismo proceso con lo que nos quedó, resto 3 = 3 * 10 = 30, lo dividimos nuevamente por 7 obteniendo cociente 4 y resto 2, el cociente es posible multiplicarlo por 0.10 y añadirlo a la parte entera quedando 7,14, esto es lo que se replica en este procedimiento en MIPS:

```
impropercase1:
        # f19 quocient
        # f8 remainder
        # f10 = 0.10
        # f11 = 0.01
        mov.s   $f1 $f9             # Multiplication parameter 1
        mov.s   $f2 $f8             # Multiplication parameter 2
        mov.s   $f8 $f24            # Empty $f8
        jal     floatMultiplication

        mov.s   $f2 $f6             # Return of multiplication
        jal     integerDivision
        add.s   $f8 $f7 $f6         # Move result to $f8

        mov.s   $f1 $f10            # input parameters for multiplication
        mov.s   $f2 $f19            # input parameters for multiplication
        mov.s   $f19 $f24           # Empty $f19
        jal     floatMultiplication # obtain the unit part 0.10 * $f19
        add.s   $f30 $f30 $f1       # Add to integer part
```

```
# Repeat the procces for hundreds

mov.s   $f1 $f9             # input parameter 1 for multiplication
mov.s   $f2 $f8             # input parameter 2 for multiplication
mov.s   $f8 $f24
jal     floatMultiplication

mov.s   $f2 $f6             # Return of multiplication
jal     integerDivision

mov.s   $f1 $f11            # input parameter 1 for multiplication
mov.s   $f2 $f19            # input parameter 2 for multiplication
mov.s   $f8 $f24            # Empty $f8
jal     floatMultiplication # obtain the hundred part 0.01 * $f19
add.s   $f30 $f30 $f1       # Add to integer part
```

```
factori:
        addi    $sp, $sp, -4            # Asigns one space in the stack to store $ra value
        sw      $ra, 0($sp)             # Store $ra value
        li      $t1, 0x10010000         # Initialize memory allocation
        sw      $a0, 0($t1)             # store in memory
        lwc1    $f1, 0($t1)             # Load in $f1 parameter
        cvt.s.w $f1, $f1

        sub.s   $f2 $f1 $f21            # f2 = f1 - 1
        mov.s   $f17 $f2                # aux = $f17

        loop:
            c.eq.s      $f24 $f2        # if f2 != 0
            bc1t        exitLoopMintt   # salir
                jal         floatMultiplication
                mov.s       $f2 $f17    # return of multiplication
                sub.s       $f17 $f17 $f21  # aux (# b = b - 1)
                sub.s       $f2 $f2 $f21    # b = b - 1
            j loop                      # loop
```

Función Factorial:

Esta función implementa una versión flotante de la factorial el cual toma registros de los datos en el coprocesador1 lo único que hace es repetir "n" veces la multiplicación de un contador que al inicio n = cont y se actualiza cont = cont – 1, hasta que sea cero retornando la respuesta.

```
expo:                               # Parameters: $f12 $f13
        addi    $sp, $sp, -4        # Asigns one space in the stack to store $ra value
        sw      $ra, 0($sp)         # Store $ra value

        mov.s   $f14 $f12           # aux "a"
        mov.s   $f15 $f13           # aux "b"

        mov.s   $f1 $f12
        loopE:
            c.eq.s  $f15 $f24       # if b == 0
            bc1t        exitE       # exit
                mov.s       $f2 $f14    # inpurt for multiplication
                jal         floatMultiplication
                sub.s       $f15 $f15 $f21
            j loopE
        exitE:
            mov.s   $f28 $f3
            mov.s   $f3 $f24
            lw      $ra, 0($sp)     # Store $ra value
            sw      $zero   0($sp)
            addi    $sp, $sp, 4     # Asigns one space in the stack to store $ra value
            jr      $ra
```

Función Potencia:

Esta función implementa la potencia de dos números localizados en los registros de coprocesador, la cual es iterativa, multiplica a veces b y retorna el valor en otro registro auxiliar, requiere la implementación de la multiplicación decimal.

Taylor sin(x) orden 5

- Representa un sumando de la sumatoria planteada anteriormente, debe repetirse "n" veces.
- Se pide al usuario el número "x" para calcular la aproximación
- Finalmente se opera cada sumando y se almacena en un parámetro.

Este procedimiento se repite cada término

```
# x**3/3!
li      $v1,0           # empty return register
li      $a1, 6          # 5!

move    $a0, $s0        # load x**3
jal     division        # x**3/3!

sub.s   $f22,$f22,$f0   # x - x**3/3!
mov.s   $f0, $f29       # Empty the return register of Coproc1 division
mov.s   $f12 $f22
jal     showAnswer
```

Taylor ln(x+1) orden 11 sin loop

- Esta representación es un sumando de la sumatoria planteada anteriormente, se hace lo mismo que el sin(x) pero operando los valores de forma diferente como es la serie original, no se utiliza la función factorial, por lo tanto en esta serie es posible llegar a valores de ordenes 10 u 11 dependiendo del número en mi representación es posible llegar a esta pero solo con números menores a 3.

```
# x**2/2
li      $v1,0           # empty return register
li      $a1, 2          # 2

move    $a0, $s0        # load x**2
jal     division        # x**2/2

sub.s   $f22,$f22,$f0   # x - x**2/2
mov.s   $f0, $f29       # Empty the return register of Coproc1 division
```

Problema 4b

Para esta implementación es necesario calcular la serie de Orden 11, por lo tanto se implementan las siguientes funciones:

Taylor sin(x) orden 11

```
loopSerie:
    beq     $t3 $s6 exitT       # exit
    move    $a0 $s7             # To input "expo" function

    # Orden en #f18 -> $f13
     sw     $s7, 88($a3)        # Store " n " 84($a3)
    lwc1    $f18, 88($a3)       # Load in $f1 parameter
    cvt.s.w $f18, $f18

    mov.s   $f12 $f16           # Number f16 -> $f12
    mov.s   $f13 $f18           # Order en #f18 -> $f13
    jal     expo                # x**n
    mov.s   $f5 $f28            # move answer

    move    $a1 $s7             # move n
    jal     factori             # n!

    mov.s   $f6 $f3             # move answer to divide
    mov.s   $f3 $f24            # empty $f3

    jal     floatDivision       # invoke floatDivision

looping:
    beq     $k0 $s6 exitT
    move    $a0 $s7             # To input "expo" function

    # Orden en #f18 -> $f13
    sw      $s7, 88($a3)        # Store " n " 84($a3)
    lwc1    $f18, 88($a3)       # Load in $f1 parameter
    cvt.s.w $f18, $f18

    mov.s   $f12 $f16           # Number f16 -> $f12
    mov.s   $f13 $f18           # Order en #f18 -> $f13
    jal     expo                # x**n
    mov.s   $f5 $f28            # move answer

    mov.s   $f6 $f18            # move answer to divide
    mov.s   $f3 $f24            # empty $f3

    jal     floatDivision       # invoke floatDivision
```

Se crea un loop donde se implementa las funciones “expo” de potencia, factori “factorial” en el coprocesador y floatDivision que realiza la división en el Coprocesador.

Cada iteración retorna un mensaje por consola el cual muestra el número del orden y el resultado de la serie hasta tal orden (Ver resultados).

Taylor ln(x+1) orden 11 con loop

Se crea una operación parecida al sin(x) usando solamente la función expo y sin factorial, es esta representación es posible llegar a valores más lejanos de mayor orden con números grandes sin problemas de memoria, Repitiendo este loop se obtiene una sumatoria completa de orden 11.

Resultados

Resultados Problema 1

```
Por favor ingrese el primer entero:  5
Por favor ingrese el segundo entero:  1
El maximo es: 5
```

```
Por favor ingrese el primer entero:  -5
Por favor ingrese el segundo entero:  8
El maximo es: 8
```

Resultados Problema 2

```
li      $a0, -230       # Number 1
li      $a1, 80         # Number 2
```

$v1	3	10

```
li      $a0, 980        # Number 1
li      $a1, 80         # Number 2
```

$v1	3	20

Resultados Problema 3A

```
li      $a0 10
li      $a1, 3
```

$v1	3	30

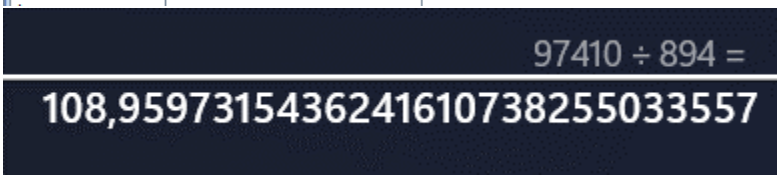
```
li      $a0 -8264
li      $a1, 7451
```

$v1	3	-61575064

Resultados Problema 3B

```
li      $a0, 97410
li      $a1, -894
```

Name	Float	Double
$f0	-108.950005	262144.19028434757




```
li      $a0, -97
li      $a1, -894
```

Name	Float	Double
$f0	0.1	0.007812501798616723

Resultados Problema 4 A-B

Taylor Series Orden 11;

Taylor sin(1) Orden11:

```
Please input a x for taylor sin(x)In range [0,1,2]: 1
 taylor sin↑1 el resultado es 0.84000003
 taylor sin↑2 el resultado es 0.84000003
 taylor sin↑3 el resultado es 0.84000003
 taylor sin↑4 el resultado es 0.84000003
 taylor sin↑5 el resultado es 0.84000003
 taylor sin↑6 el resultado es 0.84000003
 taylor sin↑7 el resultado es 0.84000003
 taylor sin↑8 el resultado es 0.84000003
 taylor sin↑9 el resultado es 0.84000003
 taylor sin↑10 el resultado es 0.84000003
 taylor sin↑11 el resultado es 0.84000003
```

Taylor sin(2) Orden 11:

```
Please input a x for taylor sin(x)In range [0,1,2]: 2
 taylor sin↑1 el resultado es 0.6700001
 taylor sin↑2 el resultado es 0.93000007
 taylor sin↑3 el resultado es 0.9100001
 taylor sin↑4 el resultado es 0.9100001
 taylor sin↑5 el resultado es 0.9100001
 taylor sin↑6 el resultado es 0.9100001
 taylor sin↑7 el resultado es 0.9100001
 taylor sin↑8 el resultado es 0.9100001
 taylor sin↑9 el resultado es 0.9100001
 taylor sin↑10 el resultado es 0.9100001
 taylor sin↑11 el resultado es 0.9100001
```

Taylor sin(3) Orden 11

```
Please input a x for taylor sin(x)In range [0,1,2]: 3
 taylor sin↑1 el resultado es -1.5
 taylor sin↑2 el resultado es 0.52
 taylor sin↑3 el resultado es 0.089999974
 taylor sin↑4 el resultado es 0.13999997
 taylor sin↑5 el resultado es 0.13999997
```

Taylor sin(4) Orden 4:

```
Please input a x for taylor sin(x)In range [0,1,2]: 4
 taylor sin↑1 el resultado es -6.660001
 taylor sin↑2 el resultado es 1.8699989
 taylor sin↑3 el resultado es -1.3800011
 taylor sin↑4 el resultado es -0.66000104
```

Nota: Mi programa se queda congelado en orden 5, pero para los casos anteriores funciona correctamente y como se puede ver la aproximación sin(3) obtenida es 0.13999 al obtenemos 2 valores iguales en **orden 4 y orden 5**

Taylor ln(4) Orden 11:

```
Please input a x for taylor ln(x+1): 4
 taylor ln(x+1) de Orden 2^-4.0
 taylor ln(x+1) de Orden 3^17.33
 taylor ln(x+1) de Orden 4^-46.67
 taylor ln(x+1) de Orden 5^158.13
 taylor ln(x+1) de Orden 6^-524.52997
 taylor ln(x+1) de Orden 7^1816.04
 taylor ln(x+1) de Orden 8^-6375.96
 taylor ln(x+1) de Orden 9^22751.148
 taylor ln(x+1) de Orden 10^-82106.45
 taylor ln(x+1) de Orden 11^299193.94
```

Taylor ln(5) Orden 11:

```
Please input a x for taylor ln(x+1): 5
 taylor ln(x+1) de Orden 2^-7.5
 taylor ln(x+1) de Orden 3^34.16
 taylor ln(x+1) de Orden 4^-122.09
 taylor ln(x+1) de Orden 5^502.91
 taylor ln(x+1) de Orden 6^-2101.2502
 taylor ln(x+1) de Orden 7^9059.46
 taylor ln(x+1) de Orden 8^-39768.66
 taylor ln(x+1) de Orden 9^177245.22
 taylor ln(x+1) de Orden 10^-799317.25
 taylor ln(x+1) de Orden 11^3396793.2
```

Taylor ln(6) Orden 10:

```
Please input a x for taylor ln(x+1): 6
 taylor ln(x+1) de Orden 2^-12.0
 taylor ln(x+1) de Orden 3^60.0
 taylor ln(x+1) de Orden 4^-264.0
 taylor ln(x+1) de Orden 5^1291.2
 taylor ln(x+1) de Orden 6^-6484.8
 taylor ln(x+1) de Orden 7^33506.05
 taylor ln(x+1) de Orden 8^-176445.95
 taylor ln(x+1) de Orden 9^943298.06
 taylor ln(x+1) de Orden 10^-5776113.0
```

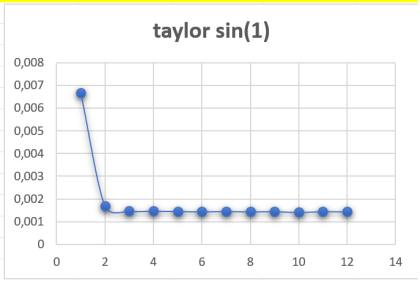
Taylor ln(3) Orden 10:

```
Please input a x for taylor ln(x+1): 3
 taylor ln(x+1) de Orden 2^-1.5
 taylor ln(x+1) de Orden 3^7.5
 taylor ln(x+1) de Orden 4^-12.75
 taylor ln(x+1) de Orden 5^35.85
 taylor ln(x+1) de Orden 6^-85.65
 taylor ln(x+1) de Orden 7^226.76999
 taylor ln(x+1) de Orden 8^-593.35
 taylor ln(x+1) de Orden 9^1593.65
 taylor ln(x+1) de Orden 10^-4311.25
 taylor ln(x+1) de Orden 11^11793.0205
```

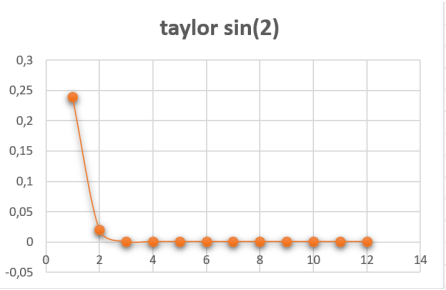
Resultados con Cálculo del Error

Sin(x)

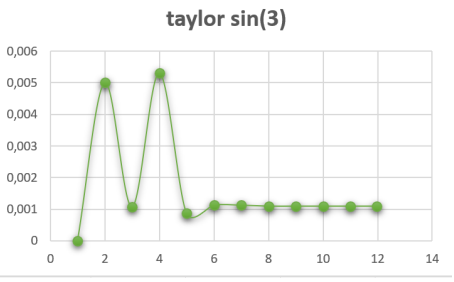
Orden	Original	Obtenido	Diferencia
1	0,83333333	0,84000003	0,0066667
2	0,84169714	0,84000003	0,00169711
3	0,84146827	0,84000003	0,00146824
4	0,84147097	0,84000003	0,00147094
5	0,84145718	0,84000003	0,00145715
6	0,84144183	0,84000003	0,0014418
7	0,84145714	0,84000003	0,00145711
8	0,84144143	0,84000003	0,0014414
9	0,84145092	0,84000003	0,00145089
10	0,841414505	0,84000003	0,001414475
11	0,8414501	0,84000003	0,00145007
12	0,84144	0,84000003	0,00143997



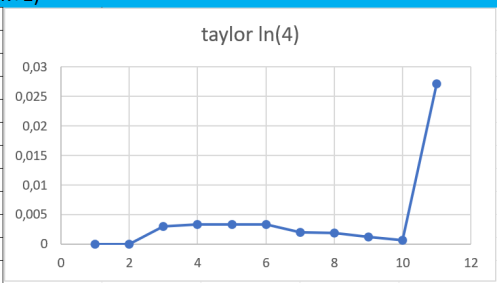
Orden	Original	Obtenido	Diferencia
1	0,9092951	0,6700001	0,239295
2	0,90929616	0,93000007	0,02070391
3	0,9092964	0,9100001	0,0007037
4	0,90929416	0,9100001	0,00070594
5	0,909296135	0,9100001	0,000703965
6	0,909296135	0,9100001	0,000703965
7	0,909295247	0,9100001	0,000704853
8	0,90929671	0,9100001	0,00070339
9	0,909296141	0,9100001	0,000703959
10	0,909296899	0,9100001	0,000703201
11	0,909296171	0,9100001	0,000703929
12	0,909296153	0,9100001	0,000703947



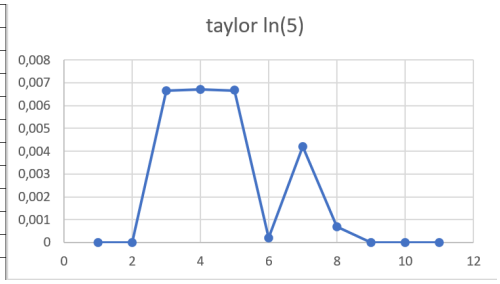
Orden	Original	Obtenido	Diferencia
1	-1,5	-1,5	0
2	0,525	0,52	0,005
3	0,09107142	0,089999974	0,001071446
4	0,1453125	0,13999997	0,00531253
5	0,14087459	0,13999997	0,00087462
6	0,141130627	0,13999997	0,001130657
7	0,141130502	0,13999997	0,001130532
8	0,1411	0,13999997	0,00110003
9	0,1411	0,13999997	0,00110003
10	0,1411	0,13999997	0,00110003
11	0,1411	0,13999997	0,00110003
12	0,1411	0,13999997	0,00110003



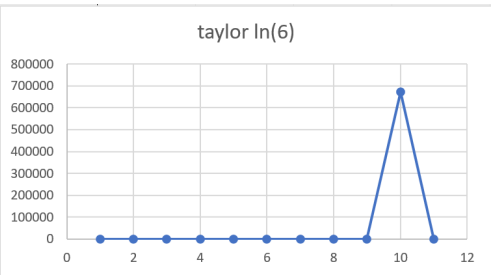
ln(x+1)			
Orden	Original	Obtenido	Diferencia
1	4	4	0
2	-4	-4	0
3	17,333	17,33	0,003
4	-46,666666	-46,67	0,003334
5	158,133333	158,13	0,003333
6	-524,533333	-524,52997	0,003363
7	1.816,04	1816,04	0,002
8	-6.375,96	-6375,96	0,0019
9	22.751,15	22751,148	0,0012
10	-82.106,45	-82106,45	0,0007
11	299.193,91	299193,94	0,02716



Orden	Original	Obtenido	Diferencia
1	5	5	0
2	-7,5	-7,5	0
3	34,16666	34,16	0,00666
4	-122,0833	-122,09	0,0067
5	502,916666	502,91	0,006666
6	-2.101,25	-2101,2502	0,0002
7	9.059,46	9059,46	0,0042
8	-39.768,66	-39768,66	0,0007
9	177245,22	177245,22	0
10	-799317,25	-799317,25	0
11	3396796,2	3396796,2	0



Orden	Original	Obtenido	Diferencia
1	6	6	0
2	-12	-12	0
3	60	60	0
4	-264	-264	0
5	1291,2	1291,2	0
6	-6484,82	-6484,8	0,02
7	33.506,06	33506,05	0,007142
8	-176.445,94	-176445,95	0,01
9	943.298,06	943298,06	0,002858
10	-5.103.319,54	-5776113	672793,4572
11			0



Análisis de gráficos:

Como se puede ver claramente existe errores muy elevados que pueden ser producto de muchos factores, la arquitectura de MIPS, errores el tipo de implementación de multiplicación, división, casos excepcionales, etc.
Parte positiva:
Los valores son extremadamente cercanos tanto que existen periodos en el gráfico donde el error tiende a 0 con una pendiente decreciente.

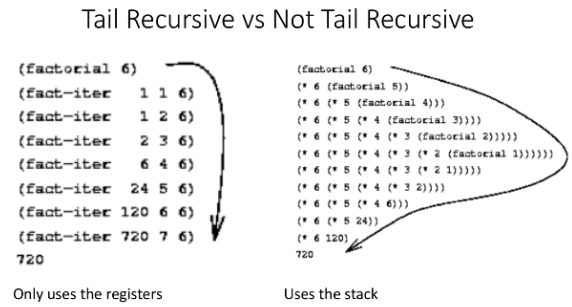
Por lo tanto se puede decir que la implementación está correcta y se puede obtener resultados reales bastante cercanos al original.

Conclusiones

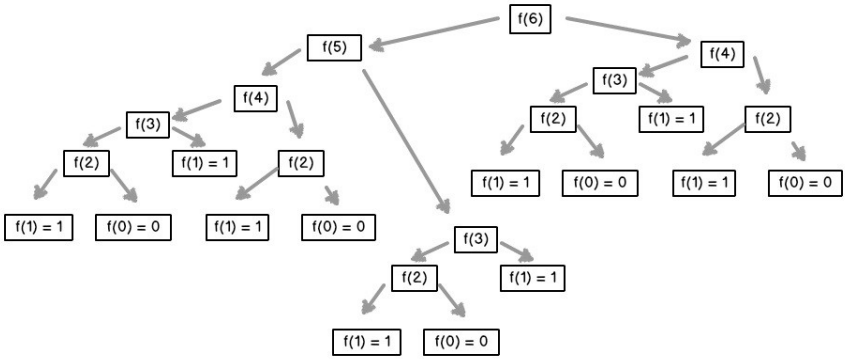
Se ha concluido este laboratorio con un gran aprendizaje en la parte de subrutinas y manejo de registros ya que es complicado preocuparse de en donde se guarda tal número o variable y si es posible poder reutilizarla correctamente, también las enseñanzas en buenas prácticas son útiles para mantener un entorno de trabajo limpio y ordenado.

Esta es la base de los computadores, la programación 100% imperativa, el programador dice donde se va a guardar tal dato y cual es el objetivo de tal procedimiento o subrutina, incluso los conceptos lejanos como el stack aquí son vistos de manera común y trabajar con estos es la mejor manera de aprender y entender como funciona en el alto nivel.

Anexo



Diferencia entre Factorial Recursivo de cola y natural



Fibonacci Recursivo Arboreo

$$\gcd\{a,b\} = \gcd\{|a|,b\} = \gcd\{a,|b|\} = \gcd\{|a|,|b|\}$$

Propiedad MCD

$$\begin{aligned}\sin(x) &= \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i+1)!} x^{2i+1} \\ &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\end{aligned}$$

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

Serie de Taylor para sin(x) y ln(1+x)