INFORME PRACTICA DE LABORATORIO 03[b] MIPS

Chillitupa Quispihuanca Alfred Addison **CCOMP3-1**

Link GitHub con los códigos

https://github.com/Alfred-Chillitupa/Laboratorio-03b-CCOMP3-1

1. **Algoritmo de ordenación por selección**

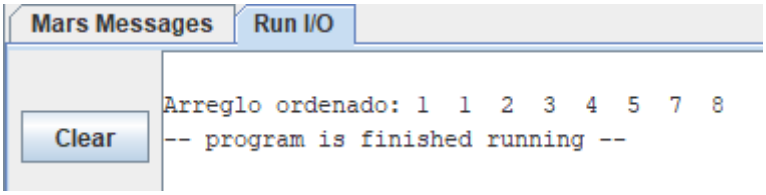
   https://github.com/Alfred-Chillitupa/Laboratorio-03b-CCOMP3-1/blob/main/selectionSortAscendente.asm

   Verifique el funcionamiento del arreglo

   Dado el siguiente arreglo desordenado

   ```
   array: .word 2,3,1,5,7,1,8,4
   ```

   La salida por consola es:

   Descripción de la Implementación

   ```
   .data
       array: .word 2,3,1,5,7,1,8,4
       strArr: .asciiz "Arreglo ordenado: "
       space: .asciiz "  "
   ```

   Iniciamos declarando nuestros datos, **array** con **.word** almacena el arreglo con los datos desordenados, seguidamente **strArr** que será para poder mostrar el resultado una ves aplicado el procedimiento, y finalmente **space** que son las separaciones de los elementos del arreglo al momento de imprimirlos.

   ```
   .text
       .globl __main

   __main:

       la $a0, array
       addi $a1,$a0,28
   ```

   En el apartado del **.text** comenzamos con un **__main** y en las siguientes líneas de código nos permiten cargar las direcciones de memoria del primer elemento

del arreglo y el ultimo elemento del arreglo en este caso el arreglo tiene 8 elementos, pero para acceder al último multiplicamos **4*7=28.**

```
max:
    li $t3, 0      # counter 0
    lw $t2, array # max
    la $v0, array # address max
    loop:
        addi $t3,$t3,4 # counter += 4
        add $t1, $a0, $t3 # INI ++

        slt $t5,$a1,$t1 # ini > FIN
        beq $t5,1,endloop

        lw $t4, 0($t1) #next element

        slt $t5, $t2, $t4 # max < ini
        beq $t5, $zero , loop #zero para ascendente

        add $t2, $t4 , 0 # max = ini
        la $v0,array+0($t3)
        j loop
    endloop:

    addi $v1,$t2,0

    jr $ra
```

El código que corresponde al procedimiento **max** desarrolla la siguiente lógica:

- ✓ Inicializamos un contador para que recorra el arreglo (como el **i++** en C), también se inicializa un registro para que almacene el valor del máximo para este caso nuestro máximo a evaluar es el primer elemento del arreglo, y en **$v0** se almacena la dirección de memoria del máximo solo que en este caso la igualamos a la dirección de memoria del primer elemento del arreglo, esto con la finalidad de realizar una operación más adelante.
- ✓ Utilizamos una sentencia de control **loop** para realizar el recorrido del arreglo y la identificación del máximo.
- ✓ Dentro de **loop** lo que se realiza es incrementar el contador en este paso **+4** porque el primer elemento ya no es necesario evaluarlo, seguidamente en **$t1** almacenamos la dirección de memoria del arreglo durante el recorrido de **array** (array + i)
- ✓ Posteriormente se define una lógica para salir del bucle y esto sucederá cuando la a causa del incremento por el contador, este sea mayor que la dirección de memoria limite del arreglo, esto con la finalidad de evaluar el último elemento de **array**.

- ✓ Cargamos el elemento de la dirección de memoria especificada **$t1** en **$t4** (arreglo[i])
- ✓ A través de una comparación se actualiza el valor del **máximo**, **debido** a que si **máximo** es menor al elemento de **$t4** quiere decir que **$t4** es el nuevo **máximo**, y si sucede esto, la dirección de memoria de ese elemento es el **array + counter** que es almacenado en **$v0**
- ✓ Finalmente retornamos en **$v1** el valor del **máximo**.

```
sort:
    beq $a0,$a1,done # single-element list is sorted
    jal max # call the max procedure
    lw $t0,0($a1) # load last element into $t0
    sw $t0,0($v0) # copy the last element to max loc
    sw $v1,0($a1) # copy max value to last element
    addi $a1,$a1,-4 # decrement pointer to last element
    j sort # repeat sort for smaller list
done:
```

Se utilizo el **sort** avanzado en clase, donde verificamos si **$a1** que es la última posición del arreglo es igual a la primera posición del arreglo, se hace la llamada al procedimiento **max,** y realizamos el intercambio del elemento **máximo** con la ultimo posición del arreglo, seguidamente decrementamos la última posición porque está ya contiene el **máximo** del arreglo.

```
la $a0, strArr
li $v0, 4
syscall
```

Se imprime en consola el **strArr** que es *"El arreglo ordenado es: "*

```
addi $t0, $zero, 0

print:
    beq $t0, 32, end
    lw $t6, array($t0)
    addi $t0, $t0, 4

    li $v0, 1
    move $a0, $t6
    syscall

    la $a0,space
    li $v0,4
    syscall

    j print
end:
```

Se recorre el arreglo manejando índices, e imprime cada elemento en consola y un **space** para separar los valores.

```
li $v0, 10
syscall
```

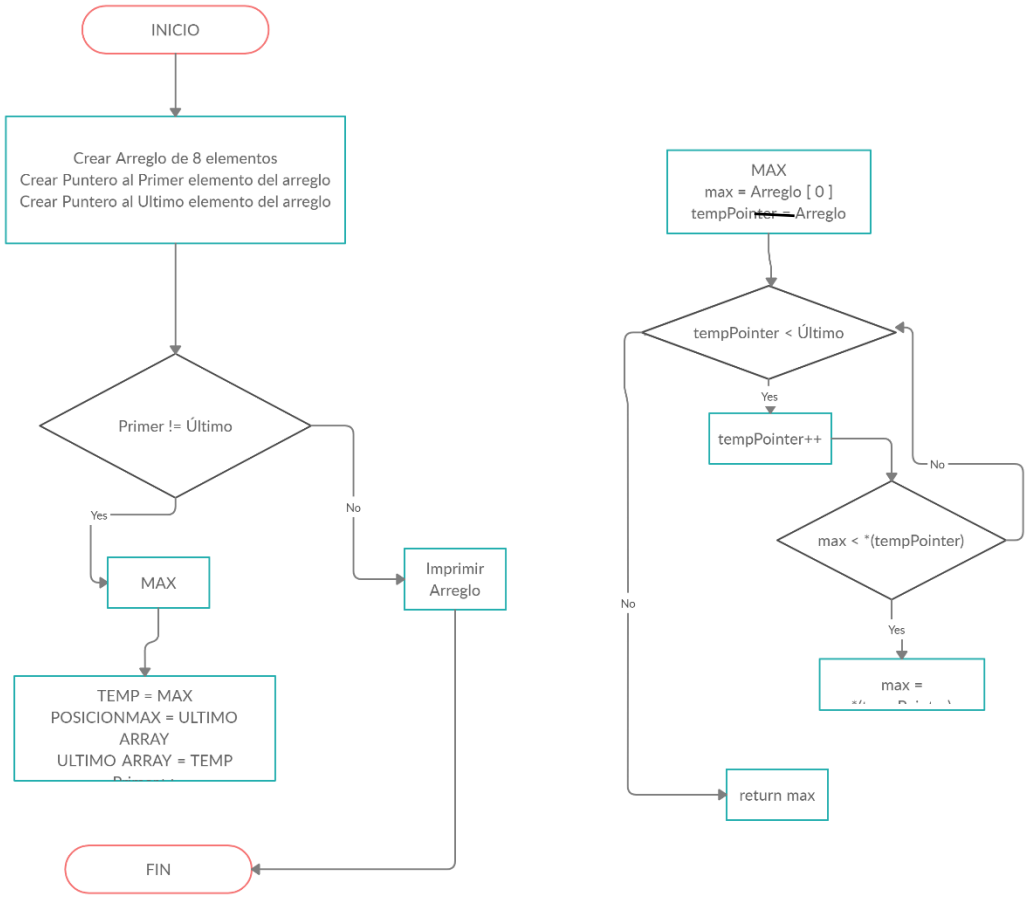
Salimos del programa y damos la ejecución como finalizada

¿Qué ocurre cuando se encuentran números repetidos?

En el procedimiento max si encuentra valores repetidos, fija el primer mayor obtenido en **max** y para la comprobación se **max** es menor al mismo numero en **max** eso es falso y el valor **max** es el primer **max** de izquierda a derecha encontrado.

En ***sort*** al fijar la última posición en **max,** esta posición ya no es considerada, y al buscar un nuevo **max**, encontrara el numero repetido y lo identificara como el nuevo **max**, es decir cada ves que reducimos el puntero al ultimo elemento generamos como sub arreglos, y el los repetidos uno de ellos se irá colocando en la última posición y uno de ellos quedara en el subarreglo convirtiéndose en el nuevo **max**.

Diagrama de flujo



Pseudo Código

```
01 INICIO Algoritmo
02     CREAR arreglo de 8 enteros ARR
03     CREAR puntero inicio del arreglo INI
04     CREAR puntero al final del arreglo FIN
05     CREAR variable mayor <- 0
06
07     FUNCION MAX
08             INI++
09             MIENTRAS (INI < FIN)
10                     IF (mayor < valor en INI)
11                             mayor <- valor en INI
12                     FIN SI
13                     INI++
14             FIN MIENTRAS
15             RETORNAR mayor
16     FIN FUNCION MAX
17
18     //SORT
19     WHILE(INI!=FIN)
20             INVOCACIÓN DE MAX
21             SWAP DE (MAYOR, FIN)
22             FIN—
23     FIN WHILE
24
25 FIN Algoritmo
```

2. **Ordenación reversa**

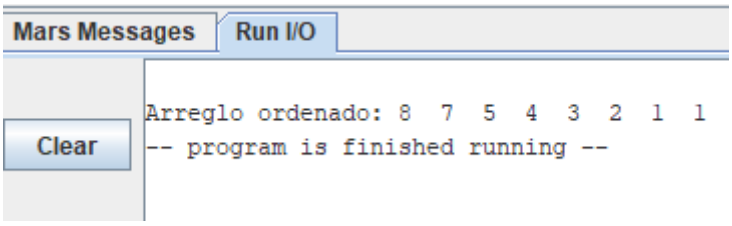
https://github.com/Alfred-Chillitupa/Laboratorio-03b-CCOMP3-1/blob/main/selectionSortDescendente.asm

Verifique el funcionamiento del arreglo

Dado el siguiente arreglo desordenado

```
array: .word 2,3,1,5,7,1,8,4
```

La salida por consola con **ordenación reversa** es:



- Para la ordenación en reversa solo es necesario cambiar la siguiente línea, resaltada de plomo de la siguiente manera,de **beq $t5, $zero , loop** a **beq $t5, 1 , loop** donde en vez de devolver el máximo devuelve el mínimo y en sort es colocado al final del arreglo

```
loop:
    addi $t3,$t3,4 # counter += 4
    add $t1, $a0, $t3 # INI ++

    slt $t5,$a1,$t1 # ini > FIN
    beq $t5,1,endloop

    lw $t4, 0($t1) #next element

    slt $t5, $t2, $t4 # max < ini
    beq $t5, 1 , loop

    add $t2, $t4 , 0 # max = ini
    la $v0,array+0($t3)
    j loop
endloop:
```

❖

3. Promedio de números enteros

- **Ejecute el programa y verifique paso a paso cuáles son los registros que se están utilizando (tanto del Procesador Principal como del CoProc01).**

Procesador Principal	

`li $s1, 11`
`la $s0, iArray`
`li $t0,0`
`li $t2,0`
`li $t3,0`
	En la parte inicial se puede apreciar los registros principales para el funcionamiento del algoritmo, **2** registros **$s** para el tamaño del arreglo y el puntero al arreglo, **3** registros **$t** que almacenarán los valores de operaciones de suma o contadores

`add $t4,$s0,$t3`
`lw $t1, 0($t4)`
`add $t0,$t0,$t1`
`addi $t2,$t2,1`
`add $t3,$0,$t2`
`add $t3,$t3,$t3`
`add $t3,$t3,$t3`
`beq $t2,$s1,endLoop`
`j loop`
	Dentro de **Loop** se pueden apreciar las operaciones que se hacen con los registros y sobre los registros, se añade dos **$t** para almacenar la posición y el valor del primer elemento del arreglo.

`la $a0, prmp01` `li $v0, 4`	En la parte final del código se hacen uso de registros del coprocesador 0 para imprimir en consola haciendo uso de **syscall**

Procesador CoProc01	

`mtc1 $t0,$f8`
`mtc1 $s1,$f9`
`cvt.s.w $f8,$f8`
`cvt.s.w $f9,$f9`
`div.s $f12,$f8,$f9`
	En esta sección del código se puede apreciar como los registros del procesador principal son movidos a los registros del coprocesador01 **$f8** y **$f9** Seguidamente se hace su transformación a punto flotante y la respectiva operación de la división para hallar el promedio

`li $v0, 2` `syscall`	En la parte final del código se hacen uso de registros del coprocesador **$f12** para imprimir en consola haciendo uso de **syscall**

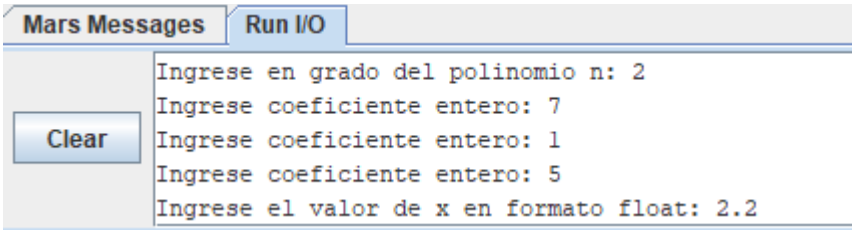
En conclusión, el código para hallar el promedio de los números, utiliza registros del procesador principal para hacer la operación de la suma de los elementos del arreglo, una vez obtenido el resultado, son enviados al coprocesador01 para su transformación y posterior operación con la división, pero estos ya no son enteros si no que son convertidos y tratados como valores en punto flotante y de esta forma se obtiene la siguiente salida por consola:

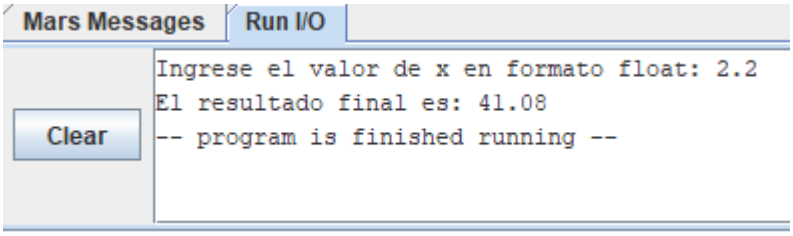
```
El promedio de los valores es: 6.0
-- program is finished running  --
```

4. Evaluación de polinomio de grado n

https://github.com/Alfred-Chillitupa/Laboratorio-03b-CCOMP3-1/blob/main/polinomio.asm

Comprobación





El algoritmo desarrollado funciona de la siguiente manera:

- En la parte de **.data** declaramos los strings necesarios para los encabezados para la interacción por consola ***iter*** que almacena las iteraciones que se necesitan para los bucles posteriores y **zero & one** son constantes 0.0 y 1.0 para trabajar e inicializar los registros.

```
.data
    str1: .asciiz "Ingrese en grado del polinomio n: "
    str2: .asciiz "Ingrese coeficiente entero: "
    str3: .asciiz "Ingrese el valor de x en formato float: "
    str4: .asciiz "El resultado final es: "
    iter: .word 0

    zero: .float 0.0
    one: .float 1.0
```

- En el **.text** definimos el **.global __main** que contiene nuestro código principal. Donde se pide el valor de **n** y luego guardamos en ***iter*** (**n+1**) y además multiplicamos 32 por **(n+1)** porque consideramos el termino independiente como un coeficiente del polinomio y reservamos memoria usando la llamada **syscall** con **$v0** = 9 y los bytes necesarios **$a0** para almacenar los coeficientes (en entero de 32 bits cada uno)

```
.text
    .globl __main

    __main:

    la $a0, str1 # Imprime "str1"
    li $v0, 4
    syscall

    li $v0, 5     # Recibe el grado del polinomio
    syscall

    addi $s0,$v0,0

    li $t0, 1
    add $t0, $t0, $s0

        sw $t0,iter

        mul $t0, $t0, 32

        li $v0, 9
        move $a0, $t0
        syscall
```

- Guardamos en **$s1** y **$s2** punteros al primer elemento del arreglo y al ultimo elemento del arreglo, y declaramos en **$t1** un puntero al primero elemento en este caso será temporal porque ayuda a recorrer el arreglo sin modificar **$s1**

```
la $s1,0($v0)#pointers
add $s2, $s1, $t0

addi $t1, $s1, 0
```

- Recorremos con los punteros el arreglo, y asignamos valores a cada dirección de memoria.

```
loop:

    beq $t1,$s2,endloop

    la $a0, str2
    li $v0, 4
    syscall

    li $v0, 5
    syscall

    move $t2,$v0
    sw $t2, 0($t1)
    addi $t1, $t1, 32
    j loop
endloop:
```

- Seguidamente cargamos los valores *float* de **zero & one,** en el Coprocesador0 y se solicita el valor de **x,** además se inicializa con un valor de 0.0 dos registros que contendrán la multiplicación de un *coeficiente por x* y *la suma de todos los términos* del polinomio (allí se almacena la respuesta)

```
lwc1 $f3, zero
lwc1 $f29,one

la $a0, str3
li $v0, 4
syscall

li $v0, 6
syscall

add.d $f4,$f0,$f4 #valor del teclado en $f4

addi $t0, $s1, 0

add.s $f7,$f7, $f3 #mult
add.s $f12,$f12, $f3 #rpta
```

- Se desarrolla un procedimiento que retorna las potencias de **x,** debido a que el termino independiente del polinomio, tiene a $\mathbf{x^0}$ que resulta **1,** utilizamos una bifurcación que evalúa el estado de nuestro exponente almacenado **$s0** para que el **$f6** (almacena el resultado de la potencia) este fijado en 1, en caso contrario realizara la potencia, utilizando la siguiente lógica: inicializamos **$f6**

con el valor de **x** y ahora solo necesitaremos hacer multiplicación de **$f6** con **x(la base), n-1** veces.

```
pow:
    li $t1, 1
    #resultado en f6
    add.s $f6,$f3,$f29
    beqz $s0,endwhile

    add.s $f6,$f3,$f3
    add.s $f6,$f4,$f3
    while:
    beq  $t1,$s0,endwhile

    mul.s $f6,$f6,$f4
    add $t1,$t1,1

    j while
    endwhile:
    add.s $f6,$f6,$f3
jr $ra
```

- Finalmente usamos otro bucle para recorrer el arreglo, y realizamos la multiplicación del primer coeficiente que se encuentra almacenado en el arreglo con su $\mathbf{x^a}$, y el valor de esta potencia se obtiene haciendo un **jal** pow, y una vez obtenido este resultado es almacenado en **$f12** para que a la segunda iteración sea actualizado sumando el nuevo valor de la multiplicación.

```
polynomial:

    beq $t0,$s2,endpolynomial

    lwc1 $f5,($t0)
    cvt.s.w $f5, $f5

    jal pow
    mul.s $f7,$f5,$f6
    add.s $f12,$f12,$f7

    addi $t0, $t0, 32
    addi $s0,$s0,-1
    j polynomial
endpolynomial:
```

- Finalmente imprimimos **str4** y como almacenamos el resultado en **$f12** podemos imprimirlo en consola y con las ultimas instrucciones se da por finalizado el programa


```
    la $a0, str4
    li $v0, 4
    syscall


    li $v0, 2
    syscall

li $v0, 10
syscall
```