

# Memoria Práctica 1

## Introducción a programación en C y lenguaje ensamblador



Guillermo del Águila Ferrandis - 100348777 - grupo 50  
guilledelaguila@gmail.com

Cristian Kublai Gómez López - 100349207 - grupo 50  
cris.kgl@gmail.com

# Índice

## *1. Parte 1. C.*

*1.1. Lectura de dimensión en línea de comandos*

*1.2. Lectura de matrices en fichero .txt (FicheroA, FicheroB)*

*1.3. Creación de la matriz C. (a partir de comparación de las matrices A y B)*

*1.4. Creación de la matriz D (constituida por valores mínimos entre los de las matrices A y B)*

*1.5. Anexo: Fallos corregidos*

## *2. Parte 2. MIPS.*

*2.1. Subrutinas pedidas en el enunciado.*

*2.1.1. minimoFloat*

*2.1.2. sumarMin*

*2.2. Subrutinas Extra*

*2.2.1. getNumByte*

*2.2.2. printMatrix*

*2.2.3. compareMatrices*

*2.2.4. cConstruction*

*2.3. Anexo: Fallos corregidos*

## *3. Banco de Errores (C y MIPS)*

# PARTE 1

## C

### 1.1.Lectura de dimensión en línea de comandos.

1.1.Filtrar la entrada para que solo se admitan números del 0 - 9

```
//to find the lenght of the 1st parameter (dimension) and check if there are any other characters other than numbers.
while ((c = argv[1][lng]) != 0){ //argv[1] = "234" => argv[1][0] = 2
    if(c < 48 || c > 57){//Only allows ascii characters 0 to 9 are allowd
        printf("Only numbers are allowed/n");
        return -1;
    }
    lng++;
}
```

1.2.Guardar en una variable “lng” la longitud del número introducido para poder realizar el paso 1.3.

```
//the value of the argument entered is obtained in its decimal value.
while(k<lng){
    int exponent = 1;
    for (int i = 1; i < (lng - k); i++){//a weight of 10^exponent is given to each character found.
        exponent = exponent * 10;
    }
    aux = aux + (argv[1][k]-'0') * exponent;
    k++;
}
```

1.3.Usar “lng” para darle peso (base 10) a cada uno de los dígitos del número introducido para obtener un valor entero.

```
//the value of the argument entered is obtained in its decimal value.
while(k<lng){
    int exponent = 1;
    for (int i = 1; i < (lng - k); i++){//a weight of 10^exponent is given to each character found.
        exponent = exponent * 10;
    }
    aux = aux + (argv[1][k]-'0') * exponent;
    k++;
}
```

1.4.Asignamos el valor entero encontrado a una variable que usaremos después y que se llamara dimensión.

## 1.2. Lectura de matrices en fichero .txt (FicheroA, FicheroB).

2.1. Guardar en una variable tipo "FILE" la dirección del fichero introducido por línea de comandos.

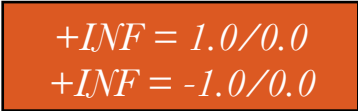
2.2. Usar la función "fscanf" con el modificador "%g" para leer los valores de la matriz almacenados en el fichero y poder así guardarlos en una matriz de tipo "float".

2.3. Mediante dos bucles "for" imprimir los valores guardados procedentes de los ficheros en las matrices de almacenamiento.

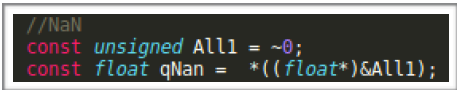
*Nota: para la escritura en fichero de las matrices C y D usaremos exactamente la misma lógica pero ahora usando "fprintf"*

## 1.3. Creación de la matriz C. (a partir de comparación de las matrices A y B)

Las primeras 3 condiciones de comparación dispuestas en el enunciado no requerían el uso de máscaras, sin embargo si necesitábamos hacer asignaciones de (+)inf, (-)inf y NaN. Para ello en los dos primeros casos simplemente usamos una comparación con una operación de floats que nos otorgaba esos valores:

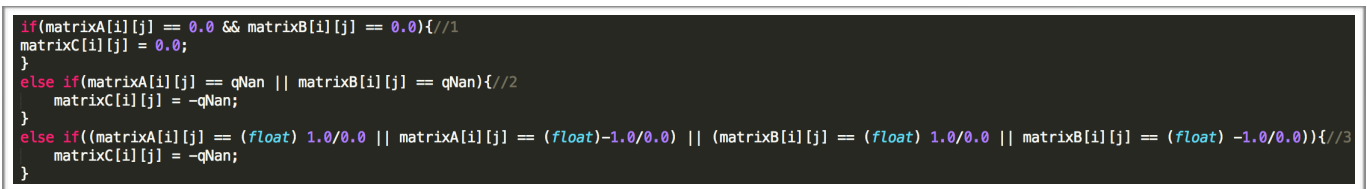

$$+INF = 1.0/0.0$$
$$-INF = -1.0/0.0$$

Para el caso de la asignación de NaN, primero probamos comparando con 0.0/0.0 pero nos salían errores en la consola que decían que el compilador no admitía la operación. Fue entonces cuando tuvimos que buscar otra manera. Finalmente tuvimos que "fabricarnos" nuestra propia constante NaN con la que pudiésemos trabajar para las comparaciones y asignaciones teniendo en cuenta que el NaN en formato IEEE 754 consta de un exponente de todo 1's y una mantisa de no todo 1's:



```
//NaN
const unsigned All1 = ~0;
const float qNaN = *((float*)&All1);
```

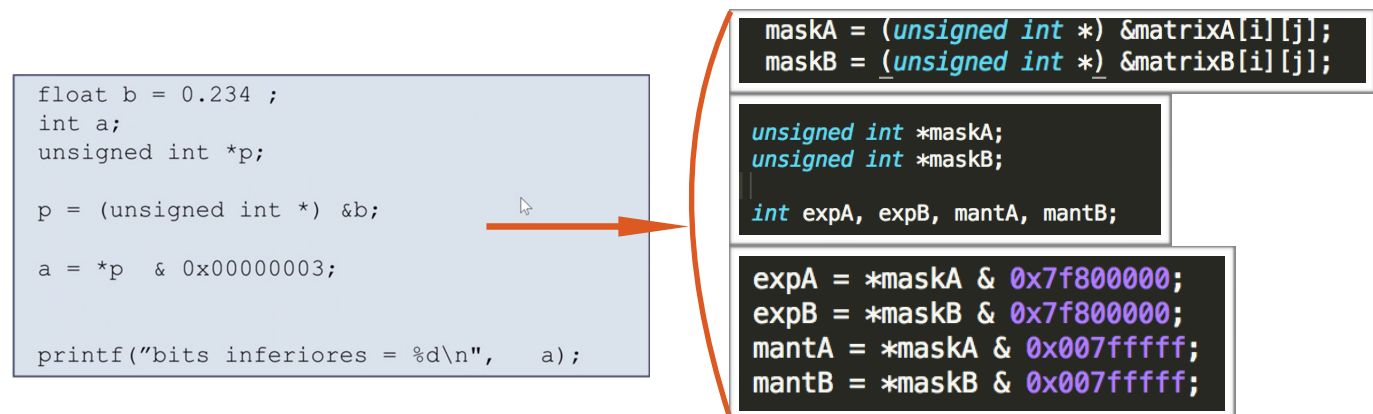
Una vez hecho esto pudimos realizar las 3 primeras comparaciones pedidas en el enunciado:



```
if(matrixA[i][j] == 0.0 && matrixB[i][j] == 0.0){//1
    matrixC[i][j] = 0.0;
}
else if(matrixA[i][j] == qNaN || matrixB[i][j] == qNaN){//2
    matrixC[i][j] = -qNaN;
}
else if((matrixA[i][j] == (float) 1.0/0.0 || matrixA[i][j] == (float)-1.0/0.0) || (matrixB[i][j] == (float) 1.0/0.0 || matrixB[i][j] == (float) -1.0/0.0)){//3
    matrixC[i][j] = -qNaN;
}
```

Para los siguientes 4 casos de comparación tuvimos que hacer uso de las máscaras para poder sustraer el exponente y la mantisa de los números en formato IEEE 754.

Nos hemos basado en las transparencias de Arcos (Intro-C):



Estábamos listos entonces para realizar las últimas 4 comparaciones:

```
else if(expA == 0 && mantA != 0 && (expB == 0) && mantB != 0){//4 A:Not normalized y B:Not normalized
    matrixC[i][j] = 0.0;
}
else if( ((expA == 0) && mantA != 0) && ((expB != 0) || (expB == 0 && mantB == 0)) ){//5:A:Not normalized y B: Normalized
    matrixC[i][j] = matrixB[i][j];
}
else if( ((expB == 0) && mantB != 0) && ((expA != 0) || (expA == 0 && mantA == 0)) ){//6:B:Not normalized y A: Normalized
    matrixC[i][j] = matrixA[i][j];
}
else if( ((expA != 0) || (expA == 0 && mantA == 0)) && ((expB != 0) || (expB == 0 && mantB == 0)) ){//7:Both Normalized
    matrixC[i][j] = (matrixA[i][j] + matrixB[i][j]);
}
```

Es importante aclarar que las comparaciones al tratarse de matrices, lo hicimos dentro de dos bucles for y aprovechamos esto para ir imprimiendo los valores que se le iban asignando a la matriz C.

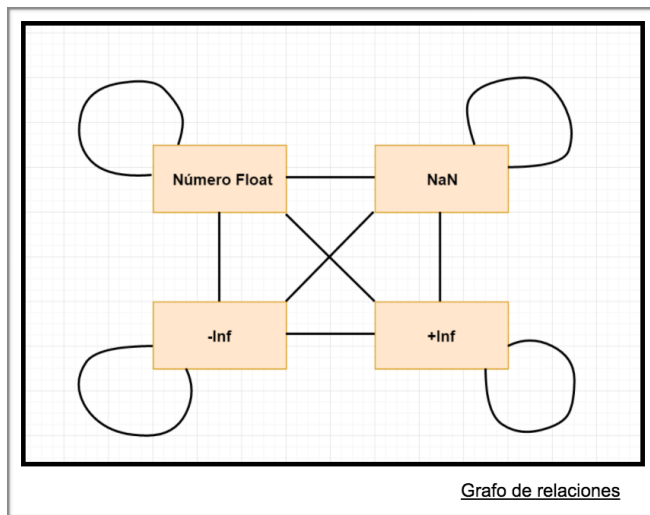
[posible error a corregir: introducir elementos distintos a los contemplados en IEEE 754]

Por último, en caso de que hubiese algún caso no contemplado al no cumplirse ninguna de las condiciones anteriores, habría que hacer saltar un mensaje de error que acabase con la construcción de la matriz C.

#### 1.4.Función para Creación de la matriz D (constituida por valores mínimos entre los de las matrices A y B).

En un primer momento teníamos la idea de realizar un diagrama de flujo que fuese excluyendo los casos más especiales de comparación ( $NaN$ ,  $+Inf$ ,  $-Inf$ ) mediante una secuencia de filtro con bloques *if*. Para ello primero tuvimos que tener en cuenta todas las posibles combinaciones y sus resultados de comparación. Lo hicimos utilizando un grafo conexo no dirigido y una tabla de resultados.

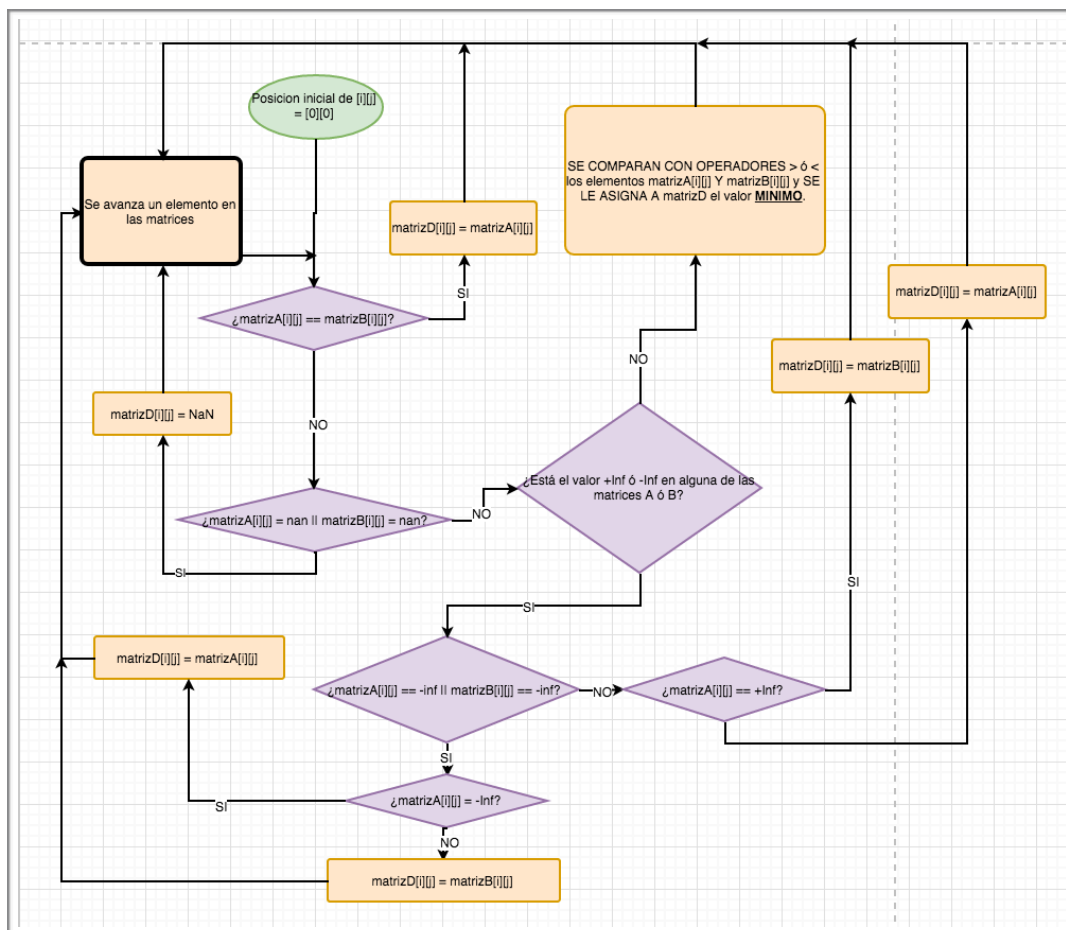
Una vez hecho esto, tuvimos que centrarnos en la lógica de filtro. Hemos querido reflejarlo mediante un diagrama de flujo para facilitar su entendimiento:



VALOR DE UNA MATRIZ	VALOR DE OTRA MATRIZ	RESULTADO
Float	Float	Mínimo valor utilizando comparación con ">" y "<"
#float	NaN	NaN
#float	+Inf	#float
#float	-Inf	-Inf
NaN	NaN	NaN
NaN	+Inf	NaN
NaN	-Inf	NaN
+Inf	+Inf	+Inf
+Inf	-Inf	-Inf
-Inf	-Inf	-Inf

Tabla de resultados

Una vez hecho esto, tuvimos que centrarnos en la lógica de filtro. Hemos querido reflejarlo mediante un diagrama de flujo para facilitar su entendimiento:



Sin embargo, nos dimos cuenta que seguro tendría que haber alguna forma más sencilla de llevar a cabo la comparación. Con lo que finalmente nos basamos en hacer una simple resta que determinase cual de los elementos era el menor (esto incluye también las comparaciones con  $(+)(-)Inf$ ).

```

void minimoFloat(int dim, float matA[dim][dim], float matB[dim][dim], float matD[dim][dim]){
    //NaN
    const unsigned All1 = ~0;
    const float qNaN = *((float*)&All1);

    int i = 0;
    int j = 0;

    for(i = 0; i < dim; i++){
        for(j = 0; j < dim; j++){
            if((matA[i][j] - matB[i][j]) > 0){
                matD[i][j] = matB[i][j];
            }else{
                matD[i][j] = matA[i][j];
            }
            if(matA[i][j] != matA[i][j]){
                matD[i][j] = -qNaN;
            }
            if(matB[i][j] != matB[i][j]){
                matD[i][j] = -qNaN;
            }
        }
    }
}

```

Después, en caso de que alguno fuese NaN, simplemente se asignaría otra vez el valor NaN a la matriz D, dejando de lado lo que se le hubiese podido asignar mediante la comparación usando la resta.

## 1.5.Anexo: Fallos corregidos

1.5.1.USO DE &—Al hacer la lectura de las matrices mediante el fscanf nos dimos cuenta que había que usar el símbolo "&" seguido de matrix[i][j](&matrix[i][j]) para poder leer el contenido de la posición de memoria matrix[i][j] y no la posición de memoria en si.

1.5.2. ERROR EN PARÉNTESIS—En la parte del código encargada de la construcción de la matriz C, tuvimos varios errores: "suggest parenthesis around && within | |". Tuvimos que ir repasando todas las condiciones y ver que ninguno de los paréntesis entrase en conflicto quedando sin cerrar.

1.5.3. DECLARACIÓN DE FUNCIONES—Es necesario definir una función antes (en el código) de llamarla. Por ejemplo no se puede definir una función en la línea 10 y llamarla en la línea 5. hay que hacerlo al revés. El mensaje de error es: "implicit declaration of function". Nos pasó al definir la función minimoFloat.

1.5.4. ERRORES EN BUCLE—Segmentation fault error: Tuvimos este error en repetidas ocasiones. Siempre acabábamos corrigiéndolo después de darnos cuenta que en algún doble bucle *for* habíamos repetido contadores, con lo que nuestros bucles acababan apuntando a direcciones de memoria en las que no había nada o había contenido que no nos interesaba.

# PARTE 2

## MIPS

Para empezar esta práctica lo primero que decidimos hacer fue aprender a recorrer matrices y a partir de ahí ir construyendo las dos subrutinas que se piden en el enunciado de la práctica.

### 2.1.SUBROUTINAS PEDIDAS EN ENUNCIADO:

#### 2.1.1.minimoFloat:

En principio creímos que la subrutina minimoFloat debería comportarse de la misma forma que el método minimoFloat de C. Luego al darnos cuenta que simplemente debía recibir dos valores *float* y dar como resultado el valor mínimo de ese par de números decidimos implementar la función

```
minimoFloat:#FUNCTION: Gives minimum from two float values///ARGUMENTS: $a2=first number; $a3=second number///RESULT:$v1

    move $t8, $a2 #saving arguments enteres in temp registers
    move $s5, $a3 #same as last line
    move $t9 $ra# $t9 Saves return value to go back to where minimoFloat was called

    blt $t8, $s5, firstLess #!$t9 < $t7? --YES-->goto: firstLess///---NO--->CONTINUE
    bgt $t8, $s5, secondLess#!$t9 > $t7? --YES-->goto: secondLess///---NO--->CONTINUE
    j areEqual
```

**2.1.2.sumarMin:** Hemos considerado dos errores que podemos detectar y dar un mensaje de alerta terminando el programa llamando a la función *exit*:

-Error 1: Comprobamos que la dimensión introducida es menor o igual a 0 usando *blez*. En caso de que lo sea, hacemos un salto a la etiqueta *errorDim1*, que cargará en *\$v1* -1, como código de error y terminará el programa haciendo un salto a la función *exit*.

-Error 2: Primero es necesario aclarar que para la detección de este error hemos usado dos etiquetas para reservar las posiciones de memoria que contienen la *dimension* y *size*. *Size* es un valor que indica la memoria necesaria para almacenar la matriz. Luego usando la subrutina *getNumByte*(Explicado en la siguiente sección Subrutinas Extra) utilizamos el valor devuelto a partir de la dimension para compararlo con *size*. Si no coincidiesen el programa simplemente ejecutaría la siguiente linea de código donde hay un salto a la etiqueta *errorDim2* donde se asigna a *\$v1* el valor -1 como código de error y se imprime por pantalla un mensaje de error asociado.

```
sumarMin: #FUNCTION: Checks if there any errors while proccessing matrix
          #ARGUMENTS: $a0 = dimension;$a1 = address of first element of matrix; $a2 address of second element of matrix
          #RESULT: $v1

#check dimension
blez $a0, errorDim1
#check dimension is coincident with array length
lw $a1, dimension
jal getNumByte #bytes needed to store all elements of matrices USING DIMENSION as an argument.
move $t7, $v1 #$t7 stores maximum value needed to sotre last element of matrix from a root direction

lw $t6, size

beq $t7, $t6, allOk
#else
j errorDim2
```



A continuación detallamos otros errores que podríamos implementar en *sumarMin*:

-Error Extra 1: Detectar si se ha introducido en las matrices cualquier otro elemento que no esté contemplado en el marco del *IEEE 754*

-Error Extra 2: Detectar que el tipo del array no es el correcto, es decir no es de tipo float o word.

-Error Extra 3: Dar un aviso por si se hubiesen introducido por equivocación una matriz de forma duplicada

## **2.2.SUBROUTINAS EXTRA:**

**2.2.1.getNumByte:** Vimos que necesitábamos tener una posición de memoria máxima para tenerlo como límite del contador que recorrería las posiciones de memoria donde estaban almacenados los contenidos de las matrices. Para ello al tratarse de matrices cuadradas, necesitábamos una función que nos pudiese dar ese número máximo de bytes ligado a la dimensión de las matrices(cuadradas).

la lógica era la siguiente:

DIMENSION = DIMENSIÓN \* DIMENSIÓN  
DIMENSIÓN = DIMENSIÓN \*4 (Bytes que ocupa cada float)

Por ejemplo, si tuviésemos una matriz 3x3. La dimensión sería 3. Tendríamos 3 x 3 elementos = 9. Cada elemento ocuparía 4 bytes con lo que el espacio necesario serían 9 x 4 = 36 Bytes.

Una vez hecho hecho, almacenaríamos el resultado obtenido en uno de los registros de resultado. Nosotros hemos usado \$v1 para evitar errores de carga en \$v0 para imprimir nuestros elementos posteriormente mediante la función printMatrix.

```
getNumByte: #Calculates number of bytes needed to store values of matrices////ARGUMENTS: needs in $a1 value of dimension////RESULT: $v1
li $t1, 4 #Size of step(4 Bytes for every element of matrix)
mul $a1, $a1, $a1 #dimension*dimension
mul $v1, $a1, $t1 #(dimension*dimension)*4
jr $ra
```

**2.2.2printMatrix:** Hemos creado una función para poder imprimir en consola el resultado de:

- MatrizA (inicializada por consola)
- MatrizB (inicializada por consola)
- MatrizD(generada usando la función *minimoFloat* y *compareMatrices*)

Primero hemos tenido que recorrer la matriz que se quisiese imprimir elemento a elemento para después hacer una llamada al sistema e imprimir el valor de almacenado cada 4 bytes contando desde el inicio de posición de memoria del array. Justamente es eso, el inicio de posición de memoria de la matriz que quisiésemos imprimir el argumento fundamental de nuestra subrutina. Es importante aclarar que hay un contador que avanza de 4 bytes en 4 bytes hasta que llega a un límite establecido por la subrutina *getNumByte* explicado anteriormente.

```
printMatrix:#FUNCTION:Prints a matrix///ARGUMENTS: address of first element of matrix=$a0///AUX: counter $t0 ///RESULT: Void
move $t1 $v1 #stores in $t1 the value of "Number of bytes needed to store all elements of matrix"
addi $t2, $zero, 4 #Sets value of step to 4

    beq $t0, $t1, endPrintMatrix

    #prints current value
    li $v0, 2
    lwcl $f12, 0($a0)
    syscall
    #STEP
    #moves to the next 4 bytes/element
    add $a0, $a0, $t2
    #Adds 4 bytes to the counter
    addi $t0, $t0, 4

    #Tab
    move $t7 $a0#storing value of address to avoid loosing it when calling the system to print a new line
    li $v0, 4
    la $a0, tabAndBreak
    syscall

    #give back old return address
    move $a0 $t7
    j printMatrix

endPrintMatrix:
    jr $ra
```

La reutilización del código es sencilla, teniendo solo que cambiar el inicio de la posición de memoria para realizar la impresión por consola de una matriz diferente.

**2.2.3.compareMatrices:** Mediante esta subrutina crearemos la matriz D siguiendo los mismos criterios de comparación que la parte de la práctica hecha en C. Para ello, a su vez dentro de esta subrutina, haremos uso de la función *minimoFloat* y de una lógica parecida a la de *printMatrix*, con la única diferencia que ahora cada vez que avancemos un elemento, guardaremos en la posición de memoria “actual” de la matriz D, el valor mínimo comparado entre la A y la B.

El principal problema de esta subrutina fue darnos cuenta de cómo debíamos tratar la existencia de infinitos y NaN's. Probamos varias cosas: primero intentamos introducir directamente el hexadecimal correspondiente a cada valor especial en las matrices sin

cambiar el tipo de dato en el *data*. Después intentamos albergar el hexadecimal en una constante cambiando el tipo de dato. Por último, y lo que nos funcionó finalmente, fue poner el hexadecimal directamente en la matriz, como al principio, pero cambiando el tipo de dato a *.word* en la inicialización de la matriz. Es importante aclarar que para hacer la asignación de

valores a la matriz D hemos usado otras subrutinas, que aunque son bastante sencillas son muy necesarias para llevar a cabo nuestra lógica de condiciones:

```
compareMatrices:#FUNCTION: Compare two float matrices element by element and assigning minimum value to same position in matrix D
#ARGUMENTS:$a1 = start address of one matrix; $a2 = start address of other matrix; $a3 = start address of matrix D
#RESULT: void
#IMPORTANT Notes: REQUIRES a pre-call to getNumByte before calling compareMatrices and move result: move $t9, $v1

move $s7, $ra#saves original main calling address in a non-temporal register
li $t0, 0 #counter = 0
move $t6, $v1 #moves result of calling getNumByte-->max-size of bytes needed to access last element of matrices

move $s1, $a1
move $s2, $a2
move $s3, $a3
#special values
#lw $t7, plusInfinity
#lw $t6, minusInfinity
lw $t5, nan

while:
    beq, $t0, $t6, endCompareMatrices #if (counter == max-size)-->goto: endCompareMatrices
    lw $t2, ($s1)#stores value of one matrix
    lw $t3, ($s2)#stores value of other matrix

    beq $t2, $t5, assignNan #if (matrix[i] == nan) -->goto: assignNan
    #else
    beq $t3, $t5, assignNan #if (matrix[i] == nan) -->goto: assignNan
    #else

    #arguments to call minimoFloat
    move $a2 $t2 # ARG: current value of one matrix
    move $a3 $t3 # ARG: current value of other matrix

    jal minimoFloat
    move $t1, $v1 #move result(minimum value found between two) to $t1
    sw $t1, ($s3)#stores minimum value found in current position of matrix D
    j step

step:
    addi $t0, $t0, 4#counter ++4
    #plus 4 bytes in all memory addresses
    addi $s1, $s1, 4
    addi $s2, $s2, 4
    addi $s3, $s3, 4
    #-----
j while
```

```
endCompareMatrices:
    move $ra, $s7#gives back return address
    jr $ra
```

```
assignNan:

    sw $t5, ($s3) #gives value NaN to matrix D
    j step
```

#### 2.2.4.cConstruction:

En esta subrutina lo que se pretende es construir la matriz C a partir de las matrices A y B, y siguiendo las condiciones presentadas en el enunciado de la práctica. Comenzamos cargando los argumentos que necesita esta subrutina. Las direcciones de inicio de las matrices A, B y C se cargan en \$a1, \$a2 y \$a3. Otros argumentos de la función son el valor de infinito, NaN y el resultado de GetNumByte. Estos tres argumentos serán importados desde la pila ya que no quedan registros \$a para entrada de argumentos.

Para poder simplificar las condiciones con los distintos signos que pueden presentar infinito y 0, lo primero que hacemos es eliminar el signo corriendo una posición a la izquierda (eliminando el signo) y otra posición a la derecha para devolver el valor a su sitio. De esta manera no nos tenemos que preocupar por el signo que tenga 0 o infinito y podemos pasar a compararlos. Las primeras condiciones que escribimos son para comprobar si el elemento en la i-ésima posición de A o B es 0, Nan o infinito.

```
beq $t3, $zero, Ais0#Check if A is 0
beq $t3, $s0, AisNan#Check if A is NaN
beq $t4, $s0, BisNan#Check if B is NaN
beq $t3, $s1, AisInfinity#Check if A is Infinity
beq $t4, $s1, BisInfinity#Check if B is Infinity
```

En el caso de que A sea 0, hay que comprobar qué es B por si fuera NaN o infinito. Si B no es 0, NaN ni infinito, saltamos a comprobar si B es un numero normalizado o no.

```

Ais0:
    beq $t4, $zero, BisZero # A = 0 and B = 0 ----> C = 0
    beq $t4, $s0, BisNaN
    beq $t4, $s1, BisInfinity
    j cConstructionContinue # A = 0 and B = float ----> jump to check normalization

```

Si los elementos  $i$ -ésimos de las matrices son floats el programa habrá pasado de largo todas las condiciones anteriores y procederá a comprobar si los floats de las matrices están normalizados o no.

- Float no normalizado -> Exponente 0 y mantisa  $\neq 0$ .

Si no se cumple la condición anterior el número es normalizado. Comenzamos analizando la mantisa y el exponente del elemento de la matriz A. Usamos instrucciones srl y sll y con las que corremos los bits de un lado a otro para quedarnos con los que nos interesa. En el caso del exponente corremos un bit a la izquierda para eliminar el signo y 23 bits a la derecha para eliminar lo que haya en la mantisa. Guardamos el resultado en un registro \$t.

Para la mantisa corremos 9 bits a la izquierda para eliminar el exponente y 9 bits a la derecha para quedarnos con los bits de la mantisa.

```

#IEEE 754 criteria
#Exponent
sll $t6, $t3, 1#SIGN: shift 1 bit to the left to eliminate bit sign
srl $t6, $t6, 24 #EXPONENT:Shift 24 bits to the right to eliminate mantisa
li $s5, 127 #load 127 in register $s5
sub $t6, $t6, $s5 #subtract 127 to obtain real value of exponent and save it in $t6
#Mantisa
sll $t7, $t3, 9 #shift 9 bits to the left to eliminate sign and exponent bits
srl $t7, $t7, 9 #return to original position of mantisa but having eliminated the sign and exponent
#save value of mantisa in $s7
beq $t6, $zero, checkMantisaA #if exponent = 0 check value of mantisa
j AisNormalized #else if the exponent is not zero then the value of matrix A is normalized
#after these conditions the value 0 or 1 will have been stored in $s6
#if $s6 = 1 ----> value is normalized
#if $s6 = 0 ----> value is not normalized

```

Ahora podemos comprobar si el elemento está normalizado o no. Si lo está, guardamos un 1 en \$s6 y un 0 si no está normalizado.

A continuación se desarrolla el mismo proceso para la matriz B, guardamos un 1 en \$s7 si está normalizado y un 0 si no está normalizado. Por ejemplo:

```

BisNormal:
    li $s7, 1 #store 1 in $s7 because B is normalized and A has a valid value
    j cConstructionContinue2 #jump to start comparing the values stored in $s6 and $s7

```

Con los valores guardados en \$s6 y \$s7 podemos comparar y construir la matriz C siguiendo las condiciones del enunciado.

```

cConstructionContinue2: #compare the values at $s6 and $s7

    bgt $s6, $s7, CEEqualsA #if $s6 = 1 and $s7 = 0 the value of matrix C will be the value of A
    bgt $s7, $s6, CEEqualsB #if $s6 = 0 and $s7 = 1 the value of matrix C will be the value of B
    beqz $s6, CIsZero #if the above conditions haven't been met and value of $s6 = 0 this means $s7 is also 0
                        #if $s6 = 0 and $s7 = 0 then the value of matrix C is 0.0

    j CEEqualsAPlusB #if the above conditions haven't been met then both values in matrix A and B are normalized
                    #of A and B are normalized C = A + B

```

Estas condiciones saltan a etiquetas que asignan los valores adecuados al elemento i-ésimo de la matriz C. por ejemplo:

```
CEqualsA:
    sw $t3, ($a3) #store value of A in C
    j step #jump to print value of element of C and continue running through matrices
CEqualsB:
    sw $t4, ($a3) #store value of B in C
    j step #jump to print value of element of C and continue running through matrices
CIsZero:
    addi $a3, $a3, 0 #C = 0
    j step
CEqualsAPlusB:
    mtc1 $t3, $f3 #move value of $t3(matrix A) to $f3 at Coproc 1
    cvt.s.w $f5, $f3 #Convert Integer to Single
    mtc1 $t4, $f4 #move value of $t4(matrix B) to $f4 at Coproc 1
    cvt.s.w $f6, $f4 #Convert Integer to Single

    add.s $f2, $f4, $f3 # A + B ----> store in $f2

    s.s $f2, 0($a3) #save the result of the sum in matrix C
    j step #jump to print value of element of C and continue running through matrices
```

En la etiqueta CEqualsAPlusB hemos tenido que pasar los valores sumar floats. Hemos tomado la decisión de solo pasar los valores al Coproc 1 en esta etiqueta ya que en la única en la que hay que hacer suma de floats. Los floats sí que se pueden comparar en los registros normales \$s y \$t y por ello no los hemos comparado en el Coproc 1, de esta manera nos ahorramos tiempo y complejidad.

### 2.3. Anexo: Fallo Corregidos

PILA MIPS—Un error que es muy común es intentar sacar datos de la pila en el mismo orden que se metieron pero esto es imposible, hay que sacarlos en orden inverso debido a la forma de funcionar de la pila.

REGISTRO \$RA—Vimos que al hacer "nesting" de funciones era necesario guardar el contenido del registro \$ra en otro registro que se conservase entre llamadas para poder guardar la dirección de memoria del "caller" inicial de nuestras funciones con funciones anidadas.

BUCLES INFINITOS—Los bucles se suelen controlar comparando los contenidos de dos registros, es decir, hasta que uno no sea igual o mayor que el otro el bucle no termina. En una ocasión no nos percatamos que uno de los registros de comparación ya había sido usado antes y no lo habíamos "limpiado" antes de volver a usarlo. El registro en cuestión contenía una dirección de memoria. Las direcciones de memoria pueden ser números muy altos por lo que el bucle parecía no terminar nunca y el programa se bloqueaba. Nos dimos cuenta de esto al depurar el código mediante el software MARS.

# PARTE 3

## BANCO DE PRUEBAS

### 3.1. Banco de pruebas de C

con parámetros de entrada correctos

```
./sumar 3 ficheroA.txt ficheroB.txt ficheroC.txt ficheroD.txt

THE DIMENSION IS ----> 3
INTRODUCED MATRICES:
MATRIX A:
inf    0      inf
2.1    2.4    1.3
3.4    6.7    0
MATRIX B:
2.3    nan    nan
1.3    2.4    1.3
3.4    6.7    0
MATRIX C:
nan    nan    nan
2.1    4.8    0
6.8    13.4   0
Matrix D:
2.3    nan    nan
1.3    2.4    1.3
3.4    6.7    0
Matrices C & D have been printed in their correspondent files
PROGRAM WORKED!!!!
```

con dimension que contenga otros números que no sean del 1 al 9.

```
crislian@ubuntu:~/Desktop/Practica1/CS$ ./sumar r ficheroA.txt ficheroB.txt ficheroC.txt ficheroD.txt
Only numbers are allowed/ncrislian@ubuntu:~/Desktop/Practica1/CS$
```

con insuficientes argumentos

```
crislian@ubuntu:~/Desktop/Practica1/CS$ ./sumar 3 ficheroA.txt ficheroB.txt ficheroC.txt
You need to type 6 parameters strictly, number of program included.
```

con dimension 0 o negativa.

```
crislian@ubuntu:~/Desktop/Practica1/CS$ ./sumar 0 ficheroA.txt ficheroB.txt ficheroC.txt ficheroD.txt
Dimension must be an integer greater than 0
```

### 3.2. Banco de pruebas MIPS

con parámetros de entrada correctos

```
No errors found to do matrix A and B processing
Matrix A: |-----|Infinity|-----|0.0|-----|Infinity|-----|2.1|-----|2.4|-----|1.3|-----|3.4|-----|6.7|-----|0.0|-----|
Matrix B: |-----|2.3|-----|NaN|-----|NaN|-----|1.3|-----|2.4|-----|1.3|-----|3.4|-----|6.7|-----|0.0|-----|
Matrix C: |-----|NaN|-----|NaN|-----|NaN|-----|2.1|-----|4.8|-----|0.0|-----|6.8|-----|13.4|-----|0.0|-----|
Matrix D: |-----|2.3|-----|NaN|-----|NaN|-----|1.3|-----|2.4|-----|1.3|-----|3.4|-----|6.7|-----|0.0|-----|
PROGRAM WORKED!
-- program is finished running --
```

con dimension 0 o negativa.

```
The dimension value must be integer value greater than 0
-- program is finished running --
```

con dimension positiva pero que no coincida con el número de bytes necesario para almacenar las matrices.

```
Dimension of matrix introduced doesnt match with the declared size needed of bytes to store the elements of matrices
-- program is finished running --
```