# Assignment 1

# Assembly programming

Álvaro Marco Pérez
NIA: 100383382
Group: 88
100383382@alumnos.uc3m.es

Ramón Hernández León
NIA: 100383351
Group: 88
100383351@alumnos.uc3m.es

# Index

# Exercise 1

## Function *set*

### Algorithm

```
function(int) set(int A[][], int M, int N){
 if( M <= 0 || N <= 0 )return 1;
 int i=0, j=0;
 do
        do
               int A[i][j] == 0;
               j++;
        while(j<N-1)
        i++;
 while(i<M-1)
 return 0;
}
```

### Test cases

| Input data | Test description | Expected output | Obtained output |
|---|---|---|---|
| A [ ][ ] → with values | The matrix A has been already initialized. | 0 | 0 |
| A [ ][ ] → char | A is a character data type matrix, therefore an invalid address. | 0 | 0 |
| M → null | M is null, therefore an invalid address. | Not compiling | Not compiling |
| M → -1 | M is negative, therefore an invalid input. | -1 | -1 |
| M → 1,1 | M is float point number, therefore an invalid input. | 0 | 0 |
| M > $2^{31}$ | M is out of bounds, therefore an invalid input. | Not compiling | Not compiling |
| N → b | N is a character, therefore an invalid input. | -1 | 0 (but unexpected result) |
| N → -35 | N is negative, therefore an invalid input. | -1 | -1 |
| N > - $2^{31}$ | N is out of bounds, therefore an invalid input. | Not compiling | Not compiling |
| All values okay | Everything is expected to work. | 0 | 0 |

## Function *add*

### Algorithm

```
function(int) add(int A[][], int B[][], int C[][], int M, int N){
 if(M <= 0 || N <= 0) return -1;
 int i = 0, int j = 0;
 do
       do
             C[i][j] = A[i][j]+B[i][j]
             j++;
       while(j<N-1)
       i++;
 while (i<M-1)
 return 0;
}
```

### Test cases

| **Input data** | **Test description** | **Expected output** | **Obtained output** |
|---|---|---|---|
| C [ ][ ] → null | The memory address that describes C does not exist. | Not compiling | Not compiling |
| B[ ][ ] → char | B is a character data type matrix, therefore an invalid address. | 0 (but unexpected result) | 0 |
| A[ ][ ] → float point | B is a float point number data type matrix, therefore an invalid address. | 0 (but unexpected result) | 0 |
| M → null | M is null, therefore an invalid address. | Not compiling | Not compiling |
| M → 1,1 | M is float point number, therefore an invalid input. | -1 | 0 (but unexpected result) |
| M > $2^{31}$ | M is out of bounds, therefore an invalid input. | Not compiling | Not compiling |
| N → b | N is a character, therefore an invalid input. | -1 | 0 (but unexpected result) |
| N → -35 | N is negative, therefore an invalid input. | -1 | -1 |
| N < $-2^{31}$ | N is float point number, therefore an invalid input. | Not compiling | Not compiling |
| All values okay | Everything is expected to work. | 0 | 0 |

## Function *extractRow*

### Algorithm

```
function (int) extractRow(int A[], int B[][], int M, int N, int j){
 if(M <= 0 || N <= 0 || j<0 || j>M-1)return -1;
 int i=0;
 do
        A[i]=B[j][i];
        i++;
 while(i<N-1)
 return 0;
}
```

### Test cases

| **Input data** | **Test description** | **Expected output** | **Obtained output** |
|---|---|---|---|
| A [ ] → null | A is null, therefore an invalid address. | Not compiling | Not compiling |
| B [ ][ ] → char type | B is a character, therefore an invalid address. | 0 (but unexpected result) | 0 ( j-th row not copied) |
| M → null | M is null, therefore an invalid address. | Not compiling | Not main or function |
| j → -1 | M is negative, therefore an invalid input. | -1 | -1 |
| M → 1,1 | M is float point number, therefore an invalid input. | -1 | -1 |
| M > $2^{31}$ | M is out of bounds, therefore an invalid input. | -1 | -1 |
| N → b | N is a character, therefore an invalid input. | 0 (but wrong result | 0 ( j-th row not copied) |
| N → -35 | N is negative, therefore an invalid input. | -1 | -1 |
| All values okay | Everything is expected to work. | 0 | 0 |

## Function *moreZeros*

### Algorithm

```
function(int) moreZeros(int A[][], int B[][], int M, int N){
 if(M <= 0 || N <= 0) return -1;

 int zeroA = calcular(A[][], M, N, 0);
 int zeroB = calcular(B[][], M, N, 0);;

 if(zeroA ==-1 || zerB ==-1)return -1;
```

```
        if(zeroA < zeroB) return 1;
        else if (zeroA > zeroB) return 0;
        else return 2;
      }
```

### Test cases

| **Input data** | **Test description** | **Expected output** | **Obtained output** |
|---|---|---|---|
| A [ ][ ] → null | A is null, therefore an invalid address. | Not compiling | Not compiling |
| B [ ][ ] → char type | B is a character, therefore an invalid address. | -1 | -1 |
| M → null | M is null, therefore an invalid address. | Not compiling | Not compiling |
| M → -1 | M is negative, therefore an invalid input. | -1 | -1 |
| M → 1,1 | M is float point number, therefore an invalid input. | -1 | -1 |
| M > $2^{31}$ | M is out of bounds, therefore an invalid input. | -1 | the web browser crashed |
| N → b | N is a character, therefore an invalid input. | -1 | |
| Wrong “calcular” inputs | As the input values are not valid the whole algorithm fails. | -1 | -1 |
| All values okay | Everything is expected to work. | 0 | 0 |

# Exercise 2

## Function *extractValues*

### Algorithm

```
function (int) extractValues (int A[][], int M, int N, int V[]){
        if(M<=0 || N<=0) return -1;
        int i=0, j=0;
        do
                do
                        if(A[i][j]==0) V[0]++;
                        else if(A[i][j]==+∞) V[1]++;
                        else if(A[i][j]==-∞) V[2]++;
                        else if(A[i][j]==NaN) V[3]++;
                        else if(A[i][j]==NotNormalized) V[4]++;
                        else if(A[i][j]==Normalized) V[5]++;
```

```
            while (j<N)
        while (i<M)
        return 0;
}
```

## Test cases

| Input data | Test description | Expected output | Obtained output |
|---|---|---|---|
| A [ ][ ] → null | A is null, therefore an invalid address. | -1 | Not compiling |
| A [ ][ ] → char type | A is a character, therefore an invalid address. | -1 | 0 (but unexpected result) |
| M → null | M is null, therefore an invalid address. | -1 | Not compiling |
| M → -1 | M is negative, therefore an invalid input. | -1 | -1 |
| M → 1,1 | M is float point number, therefore an invalid input. | -1 | 0 (but unexpected result) |
| M → Out of bounds | M is out of bounds, therefore an invalid input. | Not compiling | Not compiling |
| N → null | N is null, therefore an invalid address. | -1 | Not compiling |
| N → b | N is a character, therefore an invalid input. | -1 | 0 (but unexpected result) |
| N → -35 | N is negative, therefore an invalid input. | -1 | -1 |
| N → Out of bounds | N is float point number, therefore an invalid input. | Not compiling | Not compiling |
| All values okay | Everything is expected to work. | 0 | 0 |

## Function *add*

### Algorithm

```
function(int) add(float A[][], float B[][], float C[][], int M,
int N){
        if (M <= 0 || N<=0) return -1;
        int i = 0 , j = 0;
                do
                        do
                                C[i][j] = A[i][j]+B[i][j];
                                j++;
                        while (j<=N)
                        j++;
```

```
            while(i<=M)
      return 0;
}
```

## Test cases

| Input data | Test description | Expected output | Obtained output |
|---|---|---|---|
| B [ ][ ] → null | A is null, therefore an invalid address. | Not compiling | Not compiling |
| A [ ][ ] → char type | A is a character, therefore an invalid address. | -1 | 0 (but unexpected result) |
| M → null | M is null, therefore an invalid address. | -1 | Not compiling |
| M → -1 | M is negative, therefore an invalid input. | -1 | -1 |
| M → 1,1 | M is float point number, therefore an invalid input. | -1 | 0 (but unexpected result) |
| M > $2^{31}$ | M is out of bounds, therefore an invalid input. | Not compiling | Not compiling |
| N → null | N is null, therefore an invalid address. | -1 | Not compiling |
| N → b | N is a character, therefore an invalid input. | -1 | 0 (but unexpected result) |
| N → -35 | N is negative, therefore an invalid input. | -1 | -1 |
| N < $-2^{31}$ | N is float point number, therefore an invalid input. | Not compiling | Not compiling |
| All values okay | Everything is expected to work. | 0 | 0 |

# Problems encountered

**During programming**

While doing exercise 2, we tried to use instructions “`c.gt.s`” and “`bc1t`”, and after many hours searching for documentations and making tries, we gave up on this algorithm and try other that extracts the exponents of the number. Practical sessions professor said us that float point number instructions were not implemented on CREATOR.

Also, when making use of library “`apoyo.o`” we had some run time freezings errors. We do not really know why was this as we had different behaviours when compiling with two different algorithms.

The rest of the programming process was straightforward and we did not find any remarkable problem.

**During tests**

We have observed that when performing the tests that involve mixing different data types ("`.ascii`", ".word", ".double"), we had to make use of "`.align`" instruction. In order to indicate to the CPU the number of bytes to read from each variable.

Moreover, it was unexpected that when performing tests involving "`.float`" data type the code would have compiled. Instead of this , float was read as a really large number. It was observed that when testing with different values for "`M`" or "`N`" *variables*, the number of iterations in the *loops*, was huge; we have denoted this in *Obtained output* as "0 (but unexpected result)".

Finally, regard testing "$2^{31}$" value for words, one of our computer's navigator crashed everytime time it tried to run the code.

## Conclusions

This assessment has been useful for reinforcing the knowledge about MIPS-32 architecture and how it operates with memory.

Firstly, we learned how to deal with registers and ALU. Different types of registers should be used for different purposes: $t registers for temporal values, $s registers for values saved across calls, $sp for accessing the top of the stack, $ra for returning from a function to the line where it was called, $a for passing arguments to functions.

Secondly, we learned about the memory layout (not all the memory can be used by the programmer) and the differences between each addressing mode. The CPU cannot operate directly with memory, first those need to be loaded into registers. Moreover, the size of each data type is fixed, therefore the CPU needs to know the alignment of the bytes in order to read correctly any variable.

Thirdly, we learn to use special registers like $sp and $ra. The limited number of registers make necessary to add an intermediate place between memory and the register bank. This is the purpose of the stack, to store values that can be accessed with a single register,the *stack pointer register* ($sp) which points to the last element added to it. Additionally, the return address register ($ra) is used to return to the instruction that was being executed before the function was called.

Finally, after finishing this assessment we are able of translating pseudo-code written in a high-level programming language to assembly code. The implementation of conditional structures (if-else) or loop structuctures (for, while, do-while) can be written perfectly in assembly. Normally, it takes more lines of code, nevertheless, if memory and registers are used optimally, the time needed to run it is smaller than with other high-level programming language, since assembly instructions are very close to machine instructions.