ASSIGNMENT 1
COMPUTER STRUCTURE

Introduction to Assembly Programming

Laura Belizón Merchán (100452273) & Jorge Lázaro Ruiz (100452172)
Group 121

100452273@alumnos.uc3m.es
100452172@alumnos.uc3m.es

Contents

Exercise 1

Behaviour of the program

Arraycompare

First of all, we move the input parameters from the argument (a) registers to the temporal (t) registers. Before getting into the function, we check there are no errors (N or i negative or equal to zero). If there are, the function jumps to "error" branch and returns -1 in v0, following the convention.

If there are no errors, we enter the "loop" branch, which will repeat until we reach the end of the array. To start, we calculate the addresses of the elements we need to access with the formula:

$$adress(a_{ij}) = init\_adress + (i * n + j) * p$$

Where a_{ij} is the element we want to access, i is the column of a_{ij}, n is the length of the rows, j is the row of a_{ij} and p is the length (in bits) of each element.[1]

Once we have accessed the two elements and loaded them in the a0 and a1 registers, we invoke the function cmp (from apoyo.o) to check whether the elements are equal. Depending on the output (v0) of cmp we jump to a new branch. If v0 == 1, we jump to the "match" branch, which will add 1 to the register we use to count how many times the number x is repeated. When the counter reaches the value i, we jump to the "sequence" branch, which will add 1 to the register we use to count the number of sequences. If v0 == 0, we jump to the "mismatch" branch, which will reset the counter of occurrences.

Next, once we have reached the end of the array, the function will jump to the "result" branch. In v0, a 0 will be returned indicating there were no errors. The counter of sequences will be returned in v1.

Finally, we jump to the "end" branch. To prevent the function from never stopping, we set $ra = -1 and then jump to $ra. However, to be able to reuse the code of arraycompare for the other function of the exercise, we only jump to end if there are no rows left to evaluate, as we will now further explain.

Matrixcompare

Firstly, we move the first four input parameters from the argument (a) registers to the temporal (t) registers. Since according to the convention, a function that needs more than four arguments must have the fifth onward stored in the stack, we need to clear the space for a word in the stack and then store the register s0 in the stack.

Before getting into the function, we check there are no errors (M, N or i negative or equal to zero). If there are, the function jumps to "error" branch and returns -1 in v0, following the convention.

If there are no errors, we move the relevant data for arraycompare to the a registers. We also start a countdown for the rows of the matrix and jump to the "row" branch. This branch will loop until the countdown reaches zero. While it differs from zero, we will call our previous function arraycompare. When this function reaches the "result" branch, it will only jump to the

[1] We use this formula (more complex than necessary for this function) so that we can reuse the code for matrixcompare.

“end” branch if the countdown has reached zero. Every iteration of arraycompare picks up from where the last left off, so arraycompare does not loop over the same row.

Pseudocode

Arraycompare

```
arraycompare (int A[], int N, int x, int i) {
      if (N <= 0 or i <= 0) {
            return -1
      }
      int c = 0
      int occurrences = 0
      int sequences = 0
      while (c != N) {
            bool match
            match = comp (A[c], x)
            c++
            if (match == true) {
                  occurrences++
                  if (occurrences == i) {
                        sequences++
                  }
            } elseif (match == false) {
            occurrences = 0
      }
      return (0, sequences)
}
```

Matrixcompare

```
matrixcompare (int A[][], int M, int N, int x, int i) {
      if (M <= 0 or N <= 0 or i <= 0) {
            return -1
      }
      int c = M
      int row = 0
      int s = 0
      while (c != 0) {
            s{} = s{} + arraycompare (int A[row][], int N, int x,
int i)
            c--
            row++
      }
      return s{}
}
```

Tests

Arraycompare

Input data	Test description	Expected output	Actual output
$[3,3,3,-1,3,2,3,3,3,3,10,3,9,13,3]$ $x=3, i=2, N=15$	Check functionality of function with standard inputs	(0, 2)	(0, 2)
$[3,3,3,-1,3,2,3,3,3,3,10,3,9,13,3]$ $x=3, i=0, N=15$	Check error if i = 0	-1	-1
$[3,0,3,-1,3,2,3,4,3,11,10,3,9,13,3]$ $x=3, i=2, N=15$	Check if function returns (0, 0)	(0, 0)	(0, 0)
$[0,-7,-7,7,-7,-7]$ $x=-7, i=2, N=6$	Check if function works for negative integers	(0, 2)	(0, 2)
$[0,-7,-7,7,-7,-7]$ $x=-7, i=2, N=0$	Check error if N = 0	-1	-1
$[0,-7,-7,7,-7,-7]$ $x=-7, i=2, N=4$	Check if function stops reading array	(0, 1)	(0, 1)
$[1,2,4,4,4]$ $x=4, i=-2, N=5$	Check error if i < 0	-1	-1
$[1,2,4,4,4]$ $x=4, i=3, N=5$	Check function works for i > 2	(0, 1)	(0, 1)
$[0,0,0,0,1]$ $x=0, i=4, N=9$	Check what happens if N is bigger than array	(0, 2)[2]	(0, 2)
$[8,8,8,8,8,8,8,8]$ $x=8, i=4, N=8$	Check sequence is not counted twice	(0, 1)	(0, 1)
$[6,6,6,6,6,6]$ $x=6, i=6, N=6$	Sequence spans the whole array	(0, 1)	(0, 1)

Matrixcompare

Input data	Test description	Expected output	Actual output
$\begin{bmatrix}0&1&2&3\\1&3&3&3\\4&3&3&2\\4&3&3&3\end{bmatrix}$ $x=3, i=2, N=4, M=4$	Check functionality of function with standard inputs	(0, 3)	(0, 3)
$\begin{bmatrix}0&1&2&3\\1&3&3&3\\4&3&3&2\\4&3&3&3\end{bmatrix}$ $x=3, i=3, N=4, M=4$	Check that with the same matrix, different i changes the output	(0, 2)	(0, 2)
$\begin{bmatrix}0&1&2&3\\1&3&3&3\\4&3&3&2\\4&3&3&3\end{bmatrix}$ $x=3, i=0, N=4, M=4$	Check error if i = 0	-1	-1

[2] We hypothesize the program will fill the array with zeros so that it can continue counting up to N without crashing.

$\begin{bmatrix}1 & 4 & 3\\4 & 5 & 5\\4 & 7 & 3\end{bmatrix}$ $x=4, i=2, N=3, M=3$	Check if function returns (0, 0)	(0, 0)	(0, 0)
$\begin{bmatrix}1 & -2 & -2\\4 & -3 & 1\\-2 & 2 & 2\end{bmatrix}$ $x=-2, i=2, N=3, M=3$	Check if function works for negative integers	(0, 1)	(0, 1)
$\begin{bmatrix}1 & -2 & -2\\4 & -3 & 1\\-2 & 2 & 2\end{bmatrix}$ $x=-2, i=2, N=0, M=3$	Check error if N = 0	-1	-1
$\begin{bmatrix}1 & -2 & -2\\4 & -3 & 1\\-2 & 2 & 2\end{bmatrix}$ $x=-2, i=2, N=3, M=0$	Check error if M = 0	-1	-1
$\begin{bmatrix}1 & 2 & 1 & 1\\4 & 5 & 6 & 7\\1 & 1 & 4 & 3\\1 & 1 & 1 & 1\end{bmatrix}$ $x=1, i=2, N=2, M=2$	Check if function stops reading array	(0, 1)[3]	(0, 1)
$\begin{bmatrix}4 & 4\\3 & 4\end{bmatrix}$ $x=4, i=-2, N=2, M=2$	Check error if i < 0	-1	-1
$\begin{bmatrix}2 & 3 & 5\\3 & 2 & 2\end{bmatrix}$ $x=2, i=2, N=3, M=2$	Check function works for non-square matrices when N > M	(0, 1)	(0, 1)
$\begin{bmatrix}2 & 1\\4 & 4\\3 & 4\end{bmatrix}$ $x=4, i=2, N=2, M=3$	Check function works for non-square matrices when N < M	(0, 1)	(0, 1)
$\begin{bmatrix}0 & 3\\0 & 1\\1 & 4\end{bmatrix}$ $x=0, i=3, N=3, M=3$	Check what happens if N is bigger than array	(0, 1)[4]	(0, 1)
$\begin{bmatrix}2 & 3 & 5\\3 & 2 & 2\end{bmatrix}$ $x=0, i=3, N=3, M=3$	Check what happens if M is bigger than array	(0, 1)	(0, 1)
$\begin{bmatrix}8 & 8\\8 & 8\end{bmatrix}$ $x=8, i=1, N=2, M=2$	Check sequence is not counted twice	(0, 2)	(0, 2)

Conclusions and problems encountered

Arraycompare

The first problem we encountered was finding the address of the element we wanted to access in the array. For that, we used the formula above, but, since in assembly only one operation can be computed in each instruction, we had to apply the formula step by step, which was confusing.

Furthermore, for some of the jumps, we needed branches, as assembly does not have ifs or nested calls. The problem we encountered for this was that, as we weren't using jal, we could

[3] The program reads only the first line (not a 2x2 matrix) since we are reading the first 4 adresses.

[4] We hypothesize the program will fill the matrix with zeros so that it can continue counting up to N without crashing. The first row will be (0, 3, 0), the second one (1, 1, 4) and the third one (0, 0, 0).

not save the PC value on the register ra, so we had to find a different way to go back to where we wanted: making and using more branches.

Matrixcompare

To implement matrixcompare, our first idea was to reuse arraycompare. To do this, we had to move the arguments that matrixcompare received which were relevant for arraycompare to the argument registers that arraycompare reads. Furthermore, one of these arguments was i, which according to the convention had to be stored in the stack, so we had to learn how to use it.

Originally, we had problems because using the value of M seemed to make the program think there was one extra row filled with zeros. This was fixed when we realized that the coordinates in our matrix started counting from cell (0,0), instead of (1,1), which was what we were used to. This would prove useful to fix the following problem.

The biggest challenge in this exercise arose when we realized our program was reading the matrix incorrectly. Even more, due to the convention we could not operate with register s2, which was our row counter. In the end, we managed to fix this by making some adjustments to how the parameters were received, since we discovered they were getting mixed up in the transfer between matrixcompare and arraycompare.

Exercise 2

Behaviour of the program

First of all, we move the input parameters from the argument (a) registers to the temporal (t) registers. Before doing any other task, we check there are no errors (N or M negative or equal to zero and i or j out of range). If there are, the function jumps to the "error" branch and returns -1 in v0, following the convention.

If there are no errors, we find the value of the element in the ij-th position of the matrix A and store it in a temporary register we won't modify throughout the program. Next, we enter the "loop" branch, which will repeat until we reach the end of the matrix. We calculate the addresses of the elements we need to access with the formula:

$$adress(a_{ij}) = init\_adress + (i * n + j) * p$$

Once we have accessed a cell, we extract the exponent and the mantissa through logic operations and store them in temporary registers. Then, we evaluate the value of the mantissa: if it is different from zero, we jump to the "exponent" branch. Otherwise, we move on to the next cell. Next, we evaluate the exponent.

If the exponent is equal to 0, we identify the number as not normalized and jump to the "notnormalized" branch. In this branch, we store in this memory address the value of the ij-th position of the matrix A we had previously copied.

If the exponent is equal to 255 (it is all ones), we identify it as NaN and jump to the "setzero" branch. In this branch, we overwrite this memory address with a 0. We had previously stored the value 0 in float register f16.

To do this in every element of the matrix, we go over it in a very similar way to how we did it in matrixcompare.

Once we have reached the end of the matrix, we jump to the "result" branch where we load a 0 in v0 (according to the parameter passing convention). Finally, we jump to the "end" branch, which loads a -1 in the ra register and does a jump return to it so that the program finishes.

Pseudocode

```
matrixmodify (int A[][], int M, int N, int x, int i){
      if (M <= 0 or N <= 0 or 0 <= i <= M-1 or 0 <= j <= N-1) {
            return -1
      }
      valueij = A[i][j]
      int c = 0
      while (c < M) {
            int d = 0
            while (d < N) {
                  e = exponent (c, d)
                  m = mantissa (c, d)
                  if (m != 0) {
                        if (e == 0) {
                              A[c][d] = valueij
                        } else if (e == 255) {
```

```
                                A[c][d] = 0
                            }
                        }
                        d++
                    }
                    c++
                }
                return 0
}
```

Tests

Input data	Test description	Expected output	Actual output
$\begin{bmatrix} 0x3f800000 & 0x7f808000 & 0 \\ 0x7f800002 & 0x00600000 & 0 \\ 0x00020440 & 0 & 2 \end{bmatrix}$ $M = 3, N = 3, i = 0, j = 0$	Check functionality of function with standard inputs	0	0
$\begin{bmatrix} 0x3f800000 & 0x7f808000 & 0 \\ 0x7f800002 & 0x00600000 & 0 \\ 0x00020440 & 0 & 2 \end{bmatrix}$ $M = 0, N = 3, i = 0, j = 0$	Check error if M = 0	-1	-1
$\begin{bmatrix} 0x3f800000 & 0x7f808000 & 0 \\ 0x7f800002 & 0x00600000 & 0 \\ 0x00020440 & 0 & 2 \end{bmatrix}$ $M = 3, N = 0, i = 0, j = 0$	Check error if N = 0	-1	-1
$\begin{bmatrix} 0x7f908080 & 0 & 0x00500083 \\ 0x7f900083 & 0 & 0x7fd00083 \end{bmatrix}$ $M = 3, N = 2, i = 0, j = 0$	What happens if A[i][j] is NaN[5]	0	0
$\begin{bmatrix} 0x7f908080 & 1 & 0x00500083 \\ 0x7f900083 & 0 & 0x7fd00083 \end{bmatrix}$ $M = 2, N = 2, i = 0, j = 1$	Check if function stops reading array	0	0[6]
$\begin{bmatrix} 0x7f908080 & 1 \\ 0x7f900083 & 0 \\ 0 & 0x00500683 \end{bmatrix}$ $M = 2, N = 2, i = 0, j = 1$	Check if function stops reading array	0	0[7]
$\begin{bmatrix} 0x7f908080 & 1 \\ 0x7f900083 & 0 \\ 0 & 0x00500683 \end{bmatrix}$ $M = 3, N = 2, i = -3, j = 1$	Check error if i < 0	-1	-1
$\begin{bmatrix} 0x7f908080 & 1 \\ 0x7f900083 & 0 \\ 0 & 0x00500683 \end{bmatrix}$ $M = 3, N = 2, i = 8, j = 1$	Check error if i > M-1	-1	-1
$\begin{bmatrix} 0x7f908080 & 1 \\ 0x7f900083 & 0 \\ 0 & 0x00500683 \end{bmatrix}$ $M = 3, N = 2, i = 0, j = -2$	Check error if j < 0	-1	-1

[5] The values that are not normalized change to NaN. A[i][j] changes to 0, since we stored the value previous to the changes.

[6] The last value, which is NaN, does not change to a 0, since the matrix is read as explained in footnotes (3) and (4).

[7] The program does not change the last number, as explained in the previous footnote.

$\begin{bmatrix} 0x7f908080 & 1 \\ 0x7f900083 & 0 \\ 0 & 0x00500683 \end{bmatrix}$ $M = 3, N = 2, i = 0, j = 5$	Check error if j > N-1	-1	-1
$\begin{bmatrix} 0x7f908080 & 1 & 0x00500083 \\ 0x7f900083 & 0 & 0x7fd00083 \end{bmatrix}$ $M = 2, N = 3, i = 0, j = 1$	Check function works for non-square matrices when N > M	0	0
$\begin{bmatrix} 0x7f908080 & 1 \\ 0x7f900083 & 0 \\ 0 & 0x00500683 \end{bmatrix}$ $M = 3, N = 2, i = 0, j = 1$	Check function works for non-square matrices when N < M	0	0

In every test, we checked that the values of the matrix are correctly changed in the memory. However, we noticed that when coping the value in A[i][j], the value does not appear as an integer in the memory. We believe this is due to machine errors.

Conclusions and problems encountered

The first thing we had to learn was how to use the float registers, which we were not used to using. We found that many of the instructions that we used with the integer registers could not be used with floats.

While we were developing the program, we spent a long time figuring out how to compare the content of a register with NaN. In the end we discovered we had to work with the IEEE-754 standard and split the number in exponent and mantissa. We used hexadecimal values so we could check our code with different kinds of NaN, instead of just typing “NaN” which CREATOR recognises (as infinity).

After realizing that we needed to use the exponent and the mantissa separately, we also figured out how to determine if a number was normalized or not, which we did not know how to do when we were just using the float values as is.