Martin Kruchinski Almeida

260915767

ECSE 324

Lab 1 report

**Part 1:**

**Exercise 1-**

**Introduction:**

Text

Description automatically generatedText

Description automatically generated The first problem was to create an iterative algorithm that calculates the inputted Fibonacci number the user wants to get. This algorithm had 2 special cases and a general case. If the **input = 0**, the **fib(0) = 0** and if the **input = 1** or **2, fib(1) = fib(2) = 0**. For those special cases I did a comparison of **#0** and **#2** with **n** and then branched to the corresponding case.

Fig 1: Special cases Fig 2: General cases

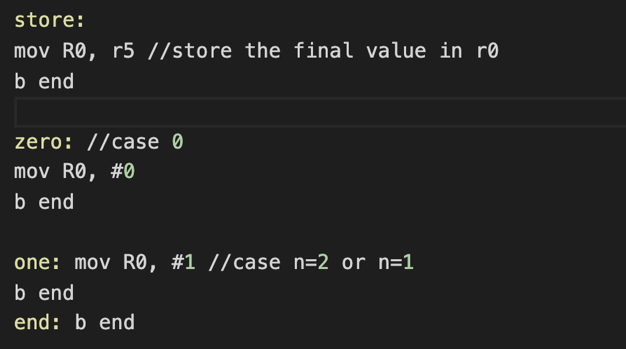


Fig 3: Dealing with special cases

For the rest of the cases I created a for loop that compares **i** with **n**. If **i** is greater than **n** the program stores the result in R0 and branches to an infinite loop, and if it is less or equal it computes **f(i) = f(i-1) + f(i-2)** , updates the values and branches to the loop again at the end. I implemented subroutines by using for loops and branches between different labels, to make the program iterative.

Challenges and improvement:

I had no challenges for this part, but I would like to improve my program by making the code shorter and simpler.

**Exercise 2-**

**Introduction**

**Text

Description automatically generated** The second problem was to create a recursive version of the Fibonacci algorithm. As the last one, this algorithm had 2 special cases and a general case. If the **input <= 2**, the algorithm would branch to a label that would take care of the special cases. Because it is a recursive algorithm, these labels were also reused by the normal cases.

**Text

Description automatically generated**

Fig 5: labels for special cases

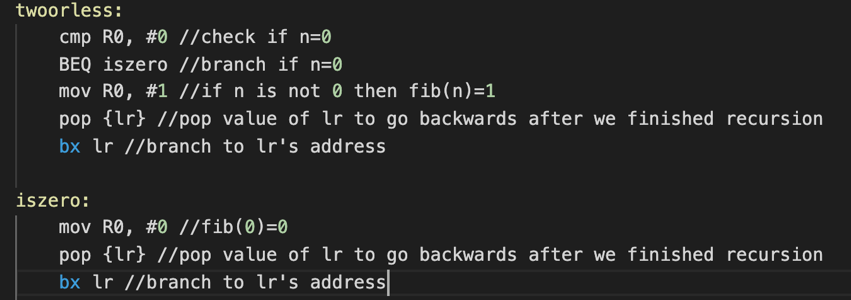
**** Fig 4: checking for special cases

Figure 6: Dealing with special cases and using stack calls

**The body of the program:**

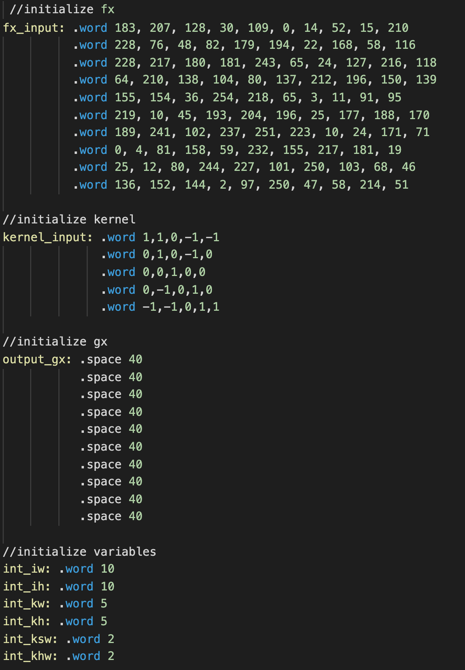
**Text

Description automatically generated** We took a subroutine and stack approach for this program. In fact, we used instructions such as **push {lr}** (that pushes the address of the next instruction into the stack), as well as **pop {register}** (that pops the value that is on top of the stack) and **bl** (that copies the address of the next instruction into the link registerand **bx** (that branches to the lr address). The reason I chose this approach was because when I did subroutines, I wanted the program to remember when I was before doing a recursive call, that is why I push the next instruction into lr. When I am done with the recursion, and I computed all the Fibonacci values, I pop those addresses that were piled in the stack to go backwards and add every Fibonacci number to compute the one we want to store in R0.

Fig 7: stack calls and subroutines instructions

Challenges and improvement:

At first, I had a hard time understanding how subroutines and stack calls were related, but later I was able to understand it. The code could be more readable.

**Part 2:**

Introduction:

In part 2 of the laboratory, I worked on a 2d convolution algorithm, which is usually used in image processing applications. The convolution algorithm is used to implement image filters and to detect objects using machine learning.

The program takes two main inputs: a 2d array **fx [10][10]** and a 2d array **kx [5][5]** called the kernel and returns a 2d array **gx[x][x]** in memory as the output. Each value in the 2d arrays is a word. For the output **gx [5][5]** a space of 40 bytes (filled with 0’s) was allocated in memory and meant to be filled once the program is done computing the algorithm. The program has also some variables like **iw, ih** (representing the width and height of the image), **kw, kh** (representing the width and height of the kernel) and **ksw, khw** (representing the Kernel width and height stride). All these variables were initialized as words. Fig 8: initialization of inputs an outputs The body of the program:

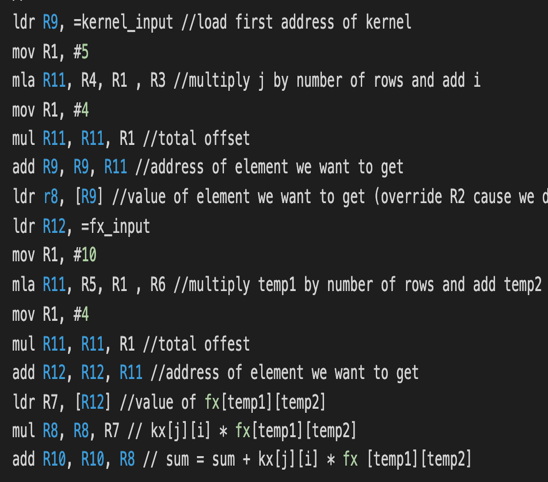
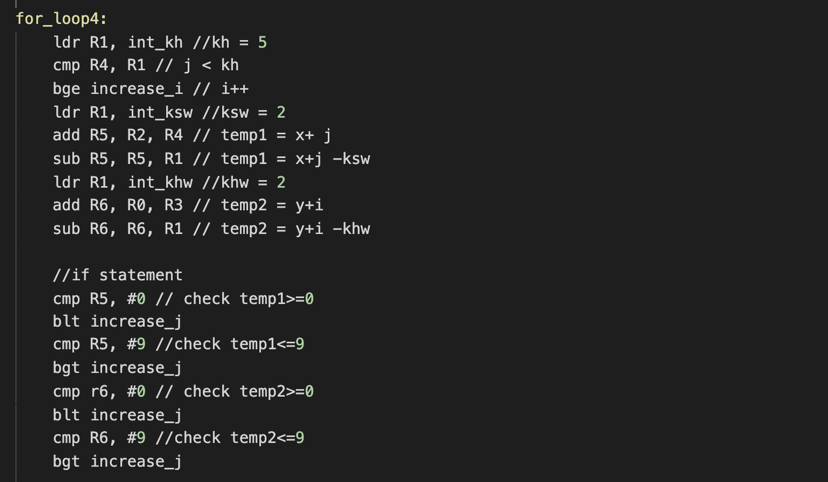
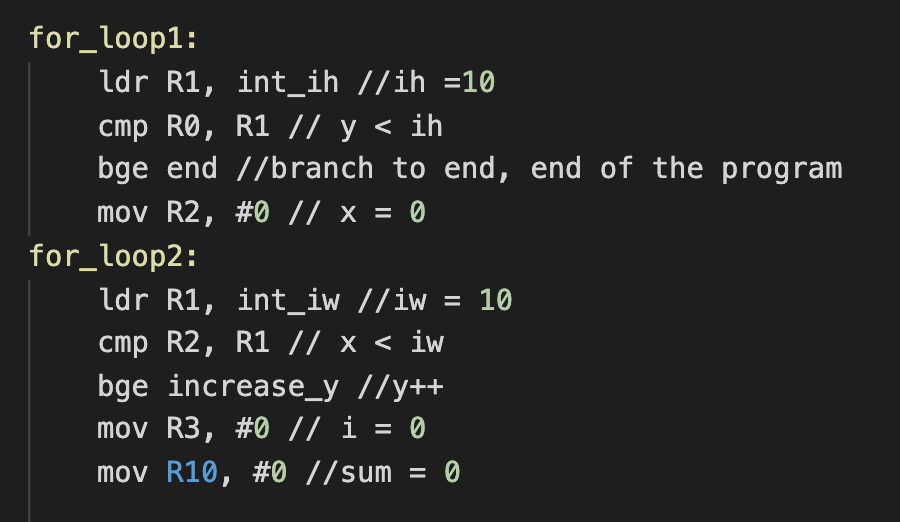
**** The body of the program consists of 4 for-loops where we compare the initial variables **iw, ih, kw and kh** to different for loop variables x, y, i, and j (also of size word). Also, the values that were going to be stored in each index of the output array, must be first stored in **sum**, a variable that would compute **sum = sum + kx[j][i] \* fx [x+j -ksw] [y+I -khw].** Before computing the sum, a series of conditions would need to be met for the sum to be finally computed. First **y < ih, x < iw, I < kw, and j < kh**, then **0 <=** x**+j -ksw <= 9, and 0 <= y+I -khw <= 9.** The reason for those values is because fx is a 10\*10 array and the indices **x+j -ksw and** y**+I -khw** need to be able to be in bounds with the array to fetch the correct value. The same reason is valid for j and i that both need to be between 0 and 4 because they are the indices of kx and 5 \* 5 array. When the conditions were not met, I implemented subroutines to increase the value of the variables and branch back to the previous loop to restart the iteration.

Figure 9: For loops Figure10: check conditions Figure11:computesum

Challenges and improvements:

The challenge I had with this program was understanding how to get the value from a specific index of a 2d array. Later, with some research, I discovered that to get the address of the index **[j][i]** of the array **kx [j] [i]** I had to compute **j \* size of array + I** and then multiply that by 4 because each word in the array takes 4 spaces in memory. I would like to make this program simpler because I think it is too long and hard to read sometimes. Also, I think it could be implemented with less computations and no need to reuse registers.

**Part 3:**

Introduction:

In the third part of my lab, I implemented the bubble sort algorithm, which sorts an input array in ascending and descending order. The user enters as input an array of size “size” and the program returns the same array in ascending or descending order.

The body:

For this algorithm, I implemented subroutines using for loops. The program has two for loops that allows to iterate through and compare the values to be sorted. The program uses pointers to point to the address in memory of the different indices of the array and load their values into registers. The ascending sorter, for example, checks if (\*(ptr + i) > \*(ptr + i + 1)) to see if the current element in the specified index is bigger than the next element in the list, and would need to be swapped.

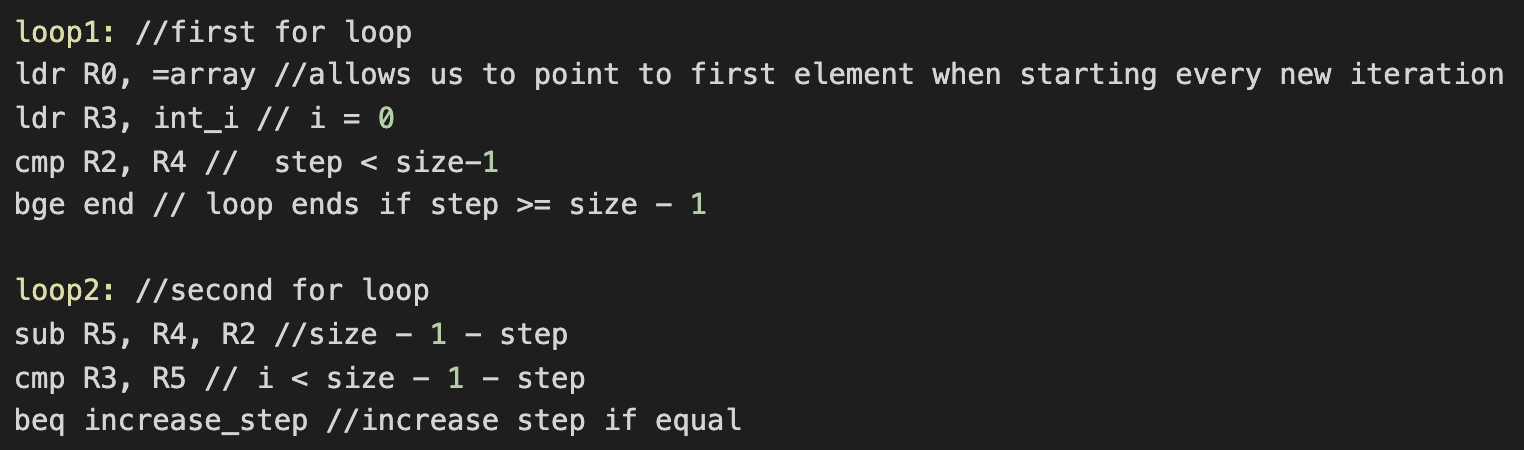


Figure 12: implementation of for loops

If the two values need to be swapped, I stored them in **registers R6 and R7**, instead of using a **temporary value** to make the swap. The value that was **in register R7** was stored **index i** and the value that was stored in register **R6** in **index i + 1** (to access the next index, I added 4 to the current address).

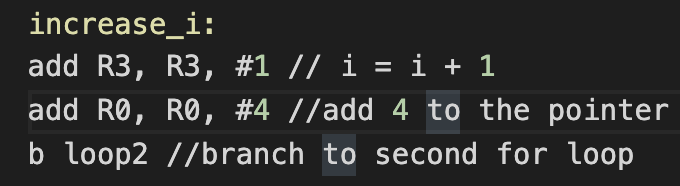
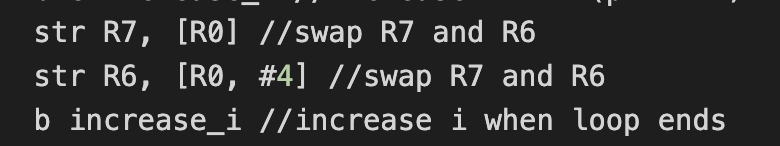
Once the values were swapped, I branched to a label to increase the for-loop variable i, incremented the value in the pointer by 4 so that we could start the comparison in the next index and branched back to the loop to continue with the iteration as we can see in figure 14.

Figure 13: swap of elements Figure 14: increase i and make pointer point to the next index

Challenges and improvements:

The only challenge I had was to understand how to use the pointers. Pointers in C have always been hard for me, but programming in Assembly, that constantly works with addresses in memories, made me have a better understanding of this concept. I would like to see if I could improve my code by making it simpler, as I use many labels that could be avoided.