ECSE 324: Lab 1

Theodore Janson, 260868223
Edin Huskic, 260795559

February 7, 2020

I. STANDARD DEVIATION

A. Problem and Approach

The problem was to find the standard deviation of a list of numbers using a shortcut involving computing the greatest and smallest numbers in the list, subtracting the greatest from the smallest and then dividing by 4. The first step was to load the relevant parameters into registers (result, pointer to the element being analyzed, number of elements in the list, maximum and minimum). The maximum and the minimum were initialized to the first element in the list.

The next step consisted in looping through every element in the list and comparing the current element to the one loaded into the max and min registers. To do this, a down counter was initialized to the list size, decremented during each loop pass and exited when its value became 0. This was done using the SUBS and BEQ instructions. At the beginning of each loop pass, the pointer was updated to the subsequent element in the list and then this value was loaded into a register. A comparison was then done between the the current element in the array and the first element in the array, which was used as a reference. During the first pass, the first element was compared to itself using instruction CMP, which raised the Z flag. The next two lines cause the code to go to new branches (BGE MAX, BGE MIN). Since the first element is equal to itself, both branches were executed. In the max branch, the current element is compared to the element stored int the max register. The current element replaces the value in the max register if it is greater than it, otherwise the program branches back to the primary loop. The min branch works similarly, but the value in the min registers is updated if the current element is less than it. In the first loop, the max and min is the first element in the array. Every element is then looped through and compared to the first element and sent to the max and min branches for comparison if greater or less than the first element. The loop exits when the counter has counted the number of elements.

The final values for the maximum and the minimum are currently in registers and the max is subtracted from the min. The difference is then right shifted by 2, which is equivalent to dividing by 4. Finally, the standard deviation is computed and stored in memory.

B. Challenges Faced

For the simplified version of the standard deviation to function, a division needed to be carried out. In this case, right shifting was the solution to achieve this. However, some testing was involved in order to identify if either arithmetic right shift or logical right shift had to be implemented. Our analysis gave positive feedback for arithmetic right shifting.

C. Possible Improvements

Possible improvements involve handling exception cases or special cases. Exception cases consist in checking for list lengths of 0 and 1, which cannot give a standard deviation. A special case could involve checking if every element in the array is equal. This could be implemented with a loop which compares the current element with the next element, and updates the current element with the next element at the end of each loop pass. The loop would exit as soon as any two consecutive elements are not equal, which realistically would be pretty early for most cases. The standard deviation could then be forced to 0. This would avoid branching to min and max branches.

II. CENTERING

A. Problem and Approach

The problem was to center an array of numbers, positive or negative. To center an array, the average of each element in the array is subtracted from each element. The first step was to load the variables (array, and array size) into registers. The sum was also initialized to 0. The next step was to add each element to the sum. To do so, we had to load a pointer for the array element being added to the sum. We then looped through each element by loading the number value of the pointer, adding it to the sum and then updating the pointer value to the next element by adding 4 to the array address each time the program looped. The loop was done using a down counter initialized to the size of the list and decremented during each pass, until it became equal to 0.

The next step was to determine the number of of times the sum should be shifted, so that the average could be computed. This required an up counter for the number of shifts and a down counter for the number of loop passes. During each loop pass, the size of the list was left shifted by 1 and the shifts were counted. This counted the number of bits after the MSB. Finally, the loop exited when the list size was shifted down to a value of 1.

Next, we had to divide the sum by the number of elements to compute the average. We used an arithmetic right shift on the sum because shifting right is equivalent to division and an arithmetic shift preserves the sign bit, which would enable the program to function with averages. We shifted with the value obtained above.

Finally, the centring was done by subtracting the newly calculated average from each element. This required another down counter and a pointer to the first element, which was updated during each loop pass. During each loop pass, a new element was loaded from the value being pointed to. The average was subtracted from it and the new value stored in memory.

B. Challenges Faced

The main challenged faced was to determine the number of shifts required to do an effective division, as we couldn’t simply shift by the number of elements. To find a solution, we started on paper with elements 1 through 8, which summed to 36 (0…100100) We needed to find the required number of shifts to turn this number into the average 4 (0…100). Evidently, we require 3 right shifts, the number of non-leading bits. Next, we needed to determine how to obtain the number 3 from 8 (0…1000) – so we right shifted 8 3 times, which was implemented by a loop which would count the number of shifts and exit when 8 became 1. This method was then tested with other values and then implemented.

C. Possible Improvements

There are some possible improvements which could have made the code faster in certain situations. We could have included a loop to detect if every element in the array was equal, in which case each element in the centred array would be 0, which could be quickly achieved. Also, checking if the array length was 0 would make the code work for more cases.

III. SORT

A. Problem and Approach

The goal of the sorting algorithm consisted of coding a program capable of filtering an input array in ascending order. For this situation, the bubble sort algorithm was chosen. Firstly, the array elements and the number of elements is both loaded into individual registers. A copy of the array is stored into another register, where most sorting operations are executed in. In order to pick a(i) and a(i-1) during the sorting iteration, two registers are used to store each element. Both registers contain the array elements and pointers, except the second register points one element further from the first register. Two other registers are used as counters for the loops used in this code.

The algorithm uses two loops to sort the array. The starting loop checks if the code iterated as many times as there are array elements by comparing the first counter with the register containing the number of elements. If the comparison results in both values being equal, the register containing the copy of the array returns its content to the original register. The new array is then stored in memory. If the counter value is less than the number of array elements, the first counter is incremented by 1 and the sorting loop is executed. For the sorting loop, each (i-1) and i elements are compared. The second counter is incremented by 1 each comparison. If the left element is greater than the right element, the element's positions are switched using an additional register. If the left element is less than the right element, the sorting loop moves on to the next elements. The sorting loop stops once i reaches the number of total elements and returns to the starting loop to recheck the first counter comparison.

B. Challenges Faced

During the coding phase, the written loops were able to iterate once through all of the array elements and behave accordingly to the bubble sort algorithm. However, a way to iterate again through the sorting loops needed to be created in order to complete the entire sorting steps. It took a considerable amount of time to adapt a complex solution, which did not end with expected results. A simpler method was chosen. The code must iterate as many times as there are array elements, guaranteeing that the output array will be sorted.

C. Possible Improvements

While the code functions as expected, it is not an efficient "bubble sort" algorithm. In fact, the code does not check the array before proceeding with the ordering loops. It simply executes as many times as there are array elements, executing as the worst-case scenario for any input array. A solution would be to add a loop that verifies if the array is already filtered before carrying on. Another concern is that the current code utilizes redundant registers. For example, two registers are used to contain the new array order, which is wasteful. A pointer to different locations of a single register would optimize the current code.

IV. Largest Integer

A. Problem and Approach

The problem was to find the greatest element in a list of numbers. The approach was to copy the code from the lab report, but also to understand it, so that sections could be used for the other coding problems. First the max register is loaded with the first element in the list. A down counter loop is used to iterated through each element. In each loop pass, the current number stored in the max register is compared to the element loaded into the "current" register. The max register is updated if the current value is greater than it. Once the counter had counted down to 0, the largest integer was stored in memory.

B. Challenges Faced

The only challenges were to not make any typos and to use proper indentation.

C. Possible Improvements

A possible improvement would be to immediately handle the case of an array size of 1 or 0. If the size is 1, the max register could be immediately forced to the value of that register. The code could also be improved by handling the case of 0 length lists.