McGill University
Computer Organization
ECSE 324

Lab #1

Emmanuelle Coutu-Nadeau (260681550)
David Kronish (260870097)

January 24, 2020

1. Finding the maximum number in a list

The first section of this lab was meant as an introduction to the assembly language. The goal of this section was to write an assembly program to find the maximum number in a given list of numbers. To achieve this goal, we were given a template code. Thus, there was no new code to write in this section of the lab.

The given code uses a loop structure in which the maximum value is stored in R0 and compared to the content of R1. R1 is updated to the next number in the list iteratively. When R1 is bigger than R0, R0 is updated to the value stored in R1. Once the counter, initially set to the number of elements in the list, reaches 0, then we have iterated over the whole list of numbers and we know that the maximal value is stored in R0.

We did not face any challenges in this section. Indeed, the code was given and worked efficiently the first time we ran it.

2. Fast standard deviation computation

The second section of this lab asked to write an assembly program to calculate the standard deviation of a list of numbers using the “range rule”. Using this method, the fast standard deviation has four main parts: finding the maximal value, finding the minimal value, a subtraction, a division.

To find the maximal value of the list, we used the same approach as section 1. To find the minimal value of the list we followed a similar approach with some minor changes. The first change is that we needed to reset the counter at the end of the LOOP_MAX block of code. Moreover, once the maximum has been found, the LOOP_MAX block of code ends by branching to the address of LOOP_MIN. LOOP_MIN uses the same structure as LOOP_MAX but uses BLE instead of BGE as in LOOP_MAX and stores the smallest value in R8. Then, to calculate the standard deviation, we subtracted the min to the max and stored it into R6 and shifted R6 twice to the left which is equivalent to dividing by 4. The standard deviation result is stored in R10.

One of the challenges we faced had to do with the use of multiple registers. As this was the first assembly program we were writing ourselves, we found that keeping up with multiple registers at the same time was challenging. Our program could be improved by using fewer registers to store values. Instead, we could reuse the same registers once they are not needed anymore.

3. Centering an array

The third section of this lab asked to make a program to easily find the center of a list of numbers. The suggested method was to calculate the average value of the list of numbers and then subtracting this value to all elements of the list. That way we can easily spot the location of the average in the list (which will be the smallest value).

To find the center of the list of numbers we decided to separate our code into 3 distinguishable blocks of code. The first loop (SUM_LOOP) adds the value of each element in the list into R6. Once the counter (R3) reaches zero, then we know that R6 contains the sum of all the elements in the list. The code then branches to CALC_AVG. This part of the code uses the sum (R6) and applies a logical right shift it by 3. We use 3 in this case because our list has 8 elements, and 8 is equivalent to 2 to the power of 3. We then reset the counter to the number of elements in the list (8) and reset the pointer to the first element in the list (R4). After this block of code, R6 contains the average value of the list. We then move into the last part: SUB_LOOP. This part of the code repeatedly subtracts the average to each element of the list until the counter reaches 0.

A challenge we faced in this section was to subtract the average to all elements including the last one. At first, we weren't able to figure out why the last loop ended before the average could be subtracted to the last element. In order to resolve this issue, we decided to use the post index mode with our counter for the SUB_LOOP. This way, the counter is only updated after the execution of the instruction and thus the last element can pass through the loop. This was very educative in understanding the effect of different addressing modes.

4. Sorting

The fourth section of this lab involved writing an assembly program to sort a list of numbers in ascending order. To achieve this goal, we decided to use a bubble sort algorithm.

There are two loops in the program and a counter which is initialized to the number of elements in the list. The OUTER_LOOP checks if the list of numbers is sorted using a "pseudo" boolean value. This is similar to a while loop. As long as the status is "unsorted" (i.e. R2 is 0) we point to the next number in the list and then go to INNER_LOOP. The INNER_LOOP compares the counter to the number of elements in the list. If the counter (R3) is equal to the number of elements, we know we have compared the current element to all other elements in the list and so we go back to the outer loop. If the counter is less, then we are not done so we compare the current value with the next value of the list. If the next number is smaller, then we store the current and next value in temporary registers and use those registers to swap the two elements without losing any information.

A challenge we faced while writing the code for this section was with branching. Indeed, to have two loops working together we really needed to think about how we branch from one loop to the other. To resolve that issue, we decided to use a register as a boolean value to track the sorting status of the list of numbers. The way this boolean register is used is similar to a while loop in Java or C. The OUTER_LOOP executes as long as the boolean register is 0 (i.e. not sorted), then the INNER_LOOP sets the boolean register to 1 (i.e. sorted), executes the swap if need be, and sets the boolean register to 0 again before recalling the OUTER_LOOP. Another solution we found to help with following the code was to separate the code into blocks having one particular function. For example, we could have done the swapping into the inner loop directly, but having a separate and identified block of code "SWAP" helped us understand what we were doing. Lastly, we were challenged by the swapping operation. After some time puzzling about how to keep the information while swapping the content of two registers we remembered a code pattern that is regularly used in C to swap two elements. This pattern uses a temporary variable to store a value while swapping the other variables. Here we used a similar pattern using registers to store the current values and then using these to swap the two elements when needed.