ECSE 324

Computer Organization

Fall 2019

Date: October 1 2019

Lab 1 Report

Group 1

Preyansh Kaushik : 260790402

Elie Elia : 2607959306

2.1 Finding a range of a set of data

This program takes in a list of n integers that are adjacent in memory and returns the range of these n values through n comparisons. The base structure of this program was provided in the lab notes, where the example given was a program to identify the maximum number in a list of elements of length n. Based on this logic of identifying the maximum number, we decided to attempt to identify the minimum number in this list as well. By doing so, we could subtract the minimum number from the maximum number to find the range of this list.

We began by initializing a register **LDR R4** to point to the location of the result from calculating the range. A register **R2** was created to hold the number of elements in our input list *n.* By doing so, the value stored in **R2** could be used as a counter for looping later in our program. Registers **R0** and **R5** were created to hold the minimum and maximum values at which the program is currently at in its sequence, but were initialized with the first element of the input list.

Next, a loop is executed with a decrementing counter on **R2** that holds the number of elements in the list (indicating the number of iterations for the loop) with a "BEQ Done" statement that jumps to the "Done" section of the program if the zero flag is set (i.e. the counter has reached the end of the list). **R3** is used to point to the next number in the list that is stored in **R1.**

A comparison is then made with the value stored in **R1** (next element in list) and **R0** (first element in list). The "BGE" logical operator follows, which checks for the result of the comparison to see if the value in **R0** is greater than the value in **R1**.If a greater value is not found, the program jumps to the "MIN" section to continue searching for the minimum value of the list. If a greater value is found, we load the value into register **R0** that holds the greatest value and continue with comparisons.

A comparison is then made to check if the current smallests element stored in **R5** is actually smaller than the value stored in **R1.** If it is not, the "BLE" logical operator directs the program to continue operating through the loop. Otherwise, the value of **R1** is stored in **R5** as the new minimum value and the looping process is repeated.

Lastly, the minimum value stored in **R0** is subtracted from the maximum value stored in **R5** in order to find the range, that is then stored in the memory location of **R0.**

Possible improvements:
Even though this provided code was efficient, concise improvements are possible. The following "DONE" loop wasn't necessary.

```
DONE:          SUBS R0, R0, R5        // subtract minimum from maximum to find range
               STR R0, [R4]           // store the result to the memory location
```

In this loop we move the contents of **R5** (containing the largest value) to R0 and then store the contents of **R0** in Result. We could have avoided this by storing the largest value directly in Result as the loop ran and found increasingly larger values. This would use less registers and remove an entire section from our code rendering it more concise.

Difficulties faced: The main roadblock we faced for this part of the assignment was getting familiar with assembly, and understanding how to work with the registers. We often confused contents of the register between memory address and value of a stored result. At the beginning, we were also unable to think of the most efficient algorithm to compute this, and tried to sort the array instead which would have required handling more registers, especially for the swapping operation in sorting. The other main issue we faced was trying to find both the maximum and the minimum of the array within the same loop. It was much easier to find the maximum and minimum in different loops, but this wouldn't be as efficient as we are iterating through the input array twice. To find both max and min in the same loop, we had to incorporate logic to jump to the "check Minimum" part of our code. We tackled this by using the "B" instruction to branch to different operations while maintaining the iteration in the same loop.

2.2 Maximum and minimum values of an algebraic expression

This program takes in a list of n + m integers that are adjacent in memory. The program then finds the maximum and minimum value of the algebraic expression of the form (x1 + x2 + x3 + . . . + xn) * (y1 + y2 + . . . + ym) using the given integers of n and m. This is done by considering all the possible combinations of n numbers and remaining m numbers and calculating their values, from which the maximum value and the minimum value can be derived.

We began by initializing a register **LDR R4** to point to the memory location of the maximum result and a register **R1** to point to the memory location of the minimum result. A register **R2** was created to hold the number of elements in our input list *n + m.* By doing so, the value stored in **R2** could be used as a counter for looping later in our program.

Register **R3** was used to point to the first number in the input list and **R4** was used to store the value in the list pointed to by **R3.** Reset **R5** to hold the sum of the values of the input elements *n + m.* Next, a loop is executed with a decrementing counter on **R2** that holds the number of elements in the list (indicating the number of iterations for the loop) with a "BEQ FOUNDSUM" statement that jumps to the "FOUNDSUM" section of the program if the zero flag is set (i.e. the counter has reached the end of the list). The value in **R4** (next element in list) is added to the sum and **R3** is used to point to the next number in the list that is stored in **R4** again**.**

Once the zero flag has been set, FOUNDSUM executes where the **R6** register is cleared and **R2** (program counter) is rest to be used for the loops later in the program. **R3** points to the next number in the list again and **R4** holds this number.

The program then starts another loop by decrementing the counter in **R2** with a "BEQ LOOPMAX" statement that jumps to the "LOOPMAX" section of the program if the zero flag is set (i.e. the counter has reached the end of the list). **R3** then points to the next number in the list, and the value is loaded into a temporary register **R10.** First number in the list at **R4** is then added to **R10** and stored in a new register **R7.** The sum of the second pair of numbers in the algebraic expression is then added by subtracting the sum of the first pair of numbers (x1+x2) from the sum of the input elements stored in **R5.**

The two sums in **R7 and R8** are now multiplied together. This multiplication result is then compared to the previously stored multiplication in **R6.** If the new multiplication is not larger, the statement "BGE LOOPMAX" jumps the program back to the loop to continue checking for multiplication of different combinations. If the new multiplied value that has been calculated is larger than the current stored maximum, the current maximum in **R6** is updated. The statement "B LOOPMAX" then jumps the program back to the loop to continue checking for the max multiplication.

After the maximum multiplication has been found, UPDATEMAX is the next step of the program where the maximum result found is stored in memory, the counter is reinitialized, R3 is pointed to the first number again and the value of this is stored in **R4**.

The program then continues onto "LOOPMIN", where the counter in **R2** is decremented to count the number of loop iterations. "BEQ UPDATEMIN" ends the loop if the counter reaches zero. **R3** points to the next number in the list and the value is stored in a temporary register **R10.** The first number of the list is then stored with the following number and stored in **R7.** This retrieves the sum of the first pair of numbers (x1+x2). The sum of the second pair of numbers in the algebraic expression is then added by subtracting the sum of the first pair of numbers (x1+x2) from the sum of the input elements stored in **R5.**

The two sums in **R7 and R8** are now multiplied together. This multiplication result is then compared to the previously stored multiplication in **R6.** If the new multiplication is not smaller, the statement "BGE LOOPMIN" jumps the program back to the loop to continue checking for multiplication of different combinations. If the new multiplied value that has been calculated is smaller than the current stored maximum, the current minimum in **R6** is updated. The statement "B LOOPMIN" then jumps the program back to the loop to continue checking for the minimum result of a multiplication. "UPDATEMIN" then stores the value of this new minimum result in memory.

Difficulties faced:

We faced a lot of issues for this part of the lab. To start with, given the problem at hand, it was quite difficult to think of even a slow algorithm that would work. We spent quite some time trying to decrypt the algorithm provided in the lab before we could begin. It became apparent that we would have to calculate the total sum of the input array, find possible results based on the provided algorithm, and keep track of the maximum and minimum of these possible results. This problem could be solved by finding all the possible combinations of summed pairs, and then checking the max and min of the result. We found this part of the problem to be particularly difficult, as most likely it would have been done recursively in a high level language, but would be immensely difficult in assembly. Another difficulty we faced with this train of thought was how to avoid choosing duplicates. For example, if you pick {1,2} as a possible pair sum, the algorithm should avoid picking {2,1} as another possible pair sum as the sum for these two numbers is already calculated.

We were unaware of how to implement something recursively on assembly, and how to solve this problem in the most optimal way, so we decided to solve the problem for N=2. This part was also difficult as there was a lot of logic that had to be implemented. The main difficulty here for us was how to reuse registers. We were both used to declaring variables freely in other programming assignments, but in assembly we had to keep track of the registers we've used and reuse them since there are a limited number of registers. We solved this issue by implementing logic to reset the registers after computing a sub-problem. For example, we reset the registers used as counters in previous loop so that they can be used as counters for the next loop.

Apart from this, we also faced a problem with how to handle the registers we use efficiently. Keeping track of the maximum and minimum required two registers on its own, but also the possible results got from our algorithm (i.e. the sum of the pairs * (Total sum of array - sum of pairs)) required another temporary register to hold this value. We tackled this by repeatedly using a single register to hold this temporary value.

Possible improvements:

There are many possible improvements to our algorithm for this part of the lab. The logic implemented has to be improved so that it can work for values that are greater than N=2. This would involve calculating sum of more than two numbers at a time. Since the problem hints at a recursive solution, a possible improvement to our assembly code would be to use a Stack to keep track of the sums at a time. We would iterate through the array, push N values onto the stack at a time, and then calculate the sum of the stack. The possible results for the maximum and minimum would then be (Sum of Stack * (Total sum of array - Sum of Stack)). This operation would repeat for every value in the array, and by the end we would have the maximum and minimum values.