McGill | Faculty of Engineering

ECSE324 – Computer Organization

Lab 1:

Introduction to ARM Programming

Group 34

Aaron Heehyun Jang – 260745290

Zheng Kun – 260579245

Submission date: 2018/02/09

1. Part1 (Maximum)

This first part was not demanding as it served as reminder of what we have learnt in class, such as the syntax, loading and storing, setting conditional flags and branching. We were also refreshed on how to use a register as a pointer and the difference between memory and resister operands.

2.1 Fast Standard Deviation Computation

To begin, we needed to decide the structure of our code, i.e., how many and where we might need labels. We were to follow the "range rule" to compute an approximation of the standard deviation.

$$\hat{\sigma} \approx \frac{x_{max} - x_{min}}{4}$$

To find the maximum and minimum element in the array, we loop through the array twice to find the maximum and minimum. The MAX_LOOP where we find the maximum is provided in Part1, and therefore reused. After successfully finding the maximum element in the array, in the MAX_DONE label, we reinitialize the pointers to the number of elements in the list and its first element and store the maximum value found in memory that we call MAX. Then, we use the same approach to find the minimum number, but instead of BGE, the BLT instruction is used to check the conditional flag after comparing two consecutive elements. After this is done, in MIN_DONE, instead of storing the minimum number into memory like we did for the maximum, it is left in the register as we need to perform operations (subtraction and division). The division by four is implemented by shifting (LSR) by 2 bits and the remainder is dropped as the quotient must be an integer.

The major problems in terms of optimization are that we loop through the identical array twice. Since we knew in advance that we would have to find the maximum and the minimum, we could have captured those values within one loop. Also, it was unnecessary to store the maximum into MAX, because it was brought back into a register later on.

2.3 Centering an Array

To compute the average, we need to add all elements in the array up and divide it by the number of elements.

Adding elements up is very easy and straight forward. The difficult part lies in how to divide the sum by the number of elements. With the assumption that the signal length is power a of two, shifting the bits to the right (LSR) was sufficient. A signal length of 8 would be represented by 1000 (last 4 digit) in binary. Since the signal length was variable, we implemented the SHIFT_COUNT where it calculates the number of right-shifts required to perform the appropriate division. We do this by LSR 8 (1000) in a for loop and have the shifted result CMP to #1, each time we shift, increment the counter by 1. By the time it hits 0001, the counter would be 3 and the CMP it to #1 and would return 0(the Condition flag to break out off the loop). Next, we subtract each element by the average number and store the result back into the same memory address we load the element from. We do this in a for loop until all the original elements have been replaced by the 'centered' version of themselves.

Potential improvement would be to use post index mode to have the pointer point towards the next element after each time the pointer has been used instead of manually increment the address that pointer register holds by #4.

2.3 Sorting

Out of many sorting algorithms, we implemented the simple bubble sort algorithm following the provided pseudo C code:

```
// Given an array A of length N
sorted = false
while not sorted:
    sorted = true
    for i = 2 to N:
        if A[i] < A[i-1], swap A[i] with A[i-1] and set sorted = false
```

Due to the lack of Booleans in Assembly, a register was used to hold the values 0 for unsorted (false) and 1 for sorted (true). Our code sets the "boolean" to true (sorted) and if it does not swap, this condition is kept true, which is the only way to exit the loop to DONE. The swap was treated separately, although it is part of the for-loop, where we would only branch to if the current element was smaller than the previous one. The swapping was done directly in memory instead of using registers to temporarily hold the elements and swapping and storing them back in memory.

The main difficulty was to keep track of the elements as they swap places and maintaining the pointer at the correct position. It was easier to take fewer elements in the array to facilitate debugging. Also, although it is a simple algorithm, it was the first time having nested loops and it took some time to grasp the concepts of branching.

Potential improvement would be to implement a faster algorithm that has a better performance than the bubble sort, such as merge sort or merge sort.