Lab1 report
Wen Cui 260824815

Within this lab, 4 algorithms are implemented. The tool used in this lab is the ARMv7 simulator website, which allowed me to write and debug assembly code.

## #1: Finding the square root of an integer number

**Description**:

The goal in this part is to implement a square root finding program using stochastic gradient descent (SGD) both iteratively and recursively. The idea was at each iteration, compute the estimate by performing step = (xi*xi-a)*xi >> k, then update the step by comparing the step with the interval bound t (which is 2 in this lab). If step is larger than 2, we update step to 2; else if step is smaller than -2, we update step to -2. After that, we update the estimate xi to xi-step. After several iterations, the estimate will be close enough to the actual value of the square root.

**Iterative Implementation:**

First of all, I initialized several registers (r0-r2) to store the estimate, counter, step value and input integer. Then I implemented a loop to update the estimate using r3 as the loop counter, which increments by 1 each iteration. For each iteration, calculate step and store in r5. By comparing r5 and 2, r5(step) is updated to different value(-2/2) by creating several branches. After updating step, the 2 branches merge together and new estimate xi=xi-step is calculated and stored in r0. We repeat the steps above until the counter reaches "cnt", then a conditional branch is used to end the program. By then, the result is stored in r0. Also, for all sections in this lab, the end branch is implemented using unconditional branch that keeps branching back to itself. In this way, random instruction execution is avoided after the termination of the program.

The main approach taken in the iterative method is branching. By checking different conditions of comparing the step value with the t, the program branches to different branches in order to update step to different value. Finally, all conditional branches are merged by an unconditional branch (SUBST) to update the estimate during each iteration. Another branch is also used to implement the loop, the loops are achieved by using a counter that was initialized to 0 and increase by 1 each iteration until it reaches count. And bge instruction is used in each iteration to check for termination of the loop. When the counter reaches count value, end branch is taken and program terminates.

**Recursive Implementation:**

To implement the algorithm recursively, the same idea of SGD was used. The main difference from the iterative approach was that instead of looping, a recursion method was used. The base case occurs when the counter is zero. In this case, the final estimate is returned and stored in r0. Otherwise, after updating the estimate as described in iterative method through branching, we call the recursive subroutine sqrtRecur on a smaller counter value, namely counter-1, by updating r2, instead of using counter for checking loop condition.

The main approach used in the recursive method is subroutine. At the start of the program, we initialized r0-r2 to store result, input integer and counter respectively as iterative method. Then we push all registers that are going to be used inside recursive subroutine call but not used outside subroutine on the stack and pop them out after the return of recursive function call. In this way, the values in these registers are restored and respect the Callee-save convention. Then the subroutine was called by bl instruction. Inside the subroutine, we update estimate (r0) the same way as iterative approach and decrease counter for each recursive call until base case is reached. In the base case, we branch back to the link register (which stores the pop {r3-lr} instruction). Then the registers r3-lr are restored. Finally, we end our program and r0 is used to store the result.

**Main Challenge:**

The iterative algorithm is relative simple and most of the code can be done by translating the given C code, no big issue happened during the implementation.

When implementing the recursive approach, the main issue I faced was with the link register. At first, I forgot to push and pop link register around the subroutine call, which caused the program branched back to wrong address when function returned. Then after multiple tries and adding multiple breakpoints using the Disassembly mode to track my error, the error was detected and solved. After that, the program is able to successfully return the final estimate.

**Possible Improvements:**

For the square root program, I think there are still several improvements that can be made to optimize the memory. By combing several branches can help save some space. Since the complexity of the if-else statements, I used multiple branches to jump back and forth for different condition. However, some branches may be combined to provide a cleaner look of the program. Another possible improvement might be decreasing the counter of the program. For example, if the estimate does not change after a few iterations, we can end the iteration and return the result instead of keep looping. It will be useless to final all iterations of the loop. In this way, the total number of subroutine calls can be reduced significantly.

## #2: Calculate the norm of an array

**Description:**

As the c code provided in the lab description, the norm of an array is calculated by the following steps. First, we calculate the log2n, where n is the size of the array. To calculate log2n, I first initialized r4 to 1 and multiply it by 2. By comparing n and r4, I increment r4 by 1 in each iteration until 1<<r4 reach n, which can be achieved using loop structure described in part1 and the shift operation. Then, the loop terminates by branching and result is stored in r4. The next part is to calculate the norm using log2n. A pointer r2 is initialized to be used to access all the elements in the array. At each loop, (*ptr)*(*ptr) is calculated and added it to the sum tmp (r3). After the termination of the loop, tmp is updated by shifting it to the right by log2n (r4), which is achieved by dividing tmp by length of array, the result is stored in r3. Finally, the square root

subroutine that was implemented in part 1 was called on r3 to give the final norm of given array, the result is stored in r0 by convention.

**Main Approach:**

Pointer manipulation and subroutine call are the keys to implement this algorithm. In order to manipulate an array, a pointer r2 is used to store the address of the array elements. R2 is initialized with the address of first array element and is incremented by 4 at each iteration to point to the next element of the array, due to the fact that every array element is a word and is aligned in memory, and has a length of 4 bytes. Subroutine call is also used when calculating the square root value for tmp at the final step. By using the push-bl-pop pattern similar to part1-2, we can call the subroutine sqrt to store the norm in r0 without affecting the registers that are only used inside the subroutine. By calling "bx lr" in the end, we are able to branch back to the pop instruction and continue to execute until we reach the end of the program.

**Main challenge**:

During the implementation of this program, the main challenge I faced was using pointer to access the array. At first, I was not aware that the content of pointer r2 is the address of array elements instead of content. Therefore, instead of adding 4 to the pointer at each iteration, I added 1 each time as the C program. As a result, misaligned exception was raised every time I run the program. However, by using the step into tool and attaching breakpoints to the program in Disassembly mode, the bug was detected and the program ran correctly in the end.

**Possible Improvement:**

There might be some possible improvements in terms of reducing required space and complexity of branching inside the program. For example, the number of registers can be reduced by reusing some registers. For example, some of the registered used when calculating log2n can be reused in the sqrt algorithm. Also, by merging several branches in the square root algorithm, the performance of this program might be improved in terms of readability.

Additionally, it would be very useful to have an instruction in ARM that can divide 2 numbers instead of by shifting. Therefore, the array length does not necessarily have to be power of 2 thus reduce the complexity of the program. In this way, the calculation of log2n will not be needed and the program can be simplified significantly. I will try to implement the division method in future labs.

## #3 Center an array

**Description:**

As described in the lab manual, the main goal for this program is to center a given array by calculating the mean of the array and subtracting the average from every element in the given array. There are several steps in implementing this algorithm. First, calculate log2n the same way as shown in part2. After that, mean is calculated in a similar way as well. More specifically, a pointer is used to access each element of the array and in each iteration, sum is stored in r3. Meanwhile, a counter is used to compare with the size of array, so that the loop can be terminated. After termination, in order to calculate the mean, we shift the sum to the right by

log2n, which performs an operation that divides sum by length of array. Finally, centering a given array is performed by using a pointer to access all the elements in an array, store the address of each element. Then calculation is performed to subtract content of each element from the mean. After then, the new content is stored at the pointer address, which ensures that in the end, each element of the final array is stored at in the same location where this element in the input array was stored.

**Main Approach:**

The main approach of this program is similar to part2, especially for the first part, namely until calculating mean. More specifically, I used pointer manipulation, shift operation and use of log2n to perform division. However, there are still some additional instruction needed to implement array centering. The main difference from part2 is the use of store instruction to store results in the given address. At each iteration, the address of each element is stored in pointer r2 and the result for each element is calculated and loaded into r6. Then "str r6 [r2]" instruction is used to store result at the memory location stored in pointer. Then we increment pointer by 4 at each iteration in order to store all elements at the same location as before.

**Main challenge:**

The first part of this program is similar to part2, therefore, it was relatively easy to implement. However, it took me some time to figure out how to store the array element at its input location. After multiple tries and by using the memory tab at the simulator, which shows the content at every address, I came up with the solution described and the result is verified by checking the memory tab as well.

Another challenge I faced was manipulating arrays with negative elements. At first, I used logical shift to compute the mean of the array as part2. However, it would give me wrong answers when array contains negative element. That is because that in part2, the elements are squared so logical shift will work. However, in part3, the sum is computed therefore arithmetic shift must be used to calculate correct results.

**Possible Improvements:**

In terms of possible improvements, same as part2, it would be very useful to have an instruction in ARM that can divide 2 numbers instead of by shifting so that the program does not have restriction on array length.

## #4 selection sort algorithm

**Description:**

The main goal in this part is the implementation of selection sort algorithm. The selection sort algorithm sorts an array by repeatedly finding and updating the minimum element from the unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array, one is already sorted, the remaining is unsorted. In every iteration of selection sort, the minimum element from the unsorted subarray is picked and moved to the sorted subarray.

**Implementation in Assembly and the approaches taken:**

In order to implement this algorithm in assembly, the knowledge of subroutine, pointer, and store load instructions are used. There are several steps to perform the sort algorithm.

At first, some registers are initialized to store pointer, number of elements, 2 counters for the inner and outer loop, respectively i and j, and the current min index as well. In the outer loop for this algorithm, we compare i with n-1 and then initialize the inner loop counter j=n+1. Inside the inner loop, we first compute the tmp value and store in r7, which is *(ptr+i). The value can be computed by loading the content at address r2(pointer) + 4*i, which can be implemented using shift operation. Then we compare it with *(ptr+j) which can be computed using similar instruction and is stored in r8. The next step is to compare r7 and r8. If tmp(r7) is larger, we update the current min index by loading the value into r7; else the current min index remains the same. After updating current min index, the swap operation is also done in inner loop by unconditional branch SWAP that swaps *(ptr+i) with *(ptr+cur_min_index),. Inside the branch SWAP, two additional registers r9, r10 are used. First we load the result value at specific pointer address to r9, r10 by load instruction. Then we can store the content of r9 and r10 at swapped addresses by store instruction but with exchanged address. In this way, we can successfully store the elements in ascending order at the same input address. After the termination of the inner loop , another branch was called to increment pointer i (r4) for outer loop, then the program can unconditionally jump to outer loop for next iteration. After iterating all array elements in outer loop, the elements will be sorted in ascending order and will be stored at the same address as they were given.

**Main challenge:**

The first issue occurred with the initialization of the inner loop counter j. At first, I put the instruction that initialized the increment counter j inside the inner loop, however, after several tests, I realized that in this case, j will be the same at each iteration of the inner loop and thus the inner loop will never end. Other than this, the other issue I faced was with the swap branch. Instead of using two extra registers to store the temp result, I used one additional register like the c code at first. However, by checking the memory tab after running the program, the value at the input memory location was messed around. After multiple tries and by looking up several resources, I found the bug and the program worked in the end.

**Possible improvement:**

There might be some possible improvements for this algorithm. First of all, I used many registers to store information, maybe in the future labs, I will try to reuse some of the registers to reduce the number of registers required for selection sort. Moreover, I will alia the registers that are used in the future labs. Labeling the registers will significantly increase the readability of the program and makes it easier to debug as well.