1.1

The problem for this part of the lab was to implement the Fibonacci sequence using an iterative approach. Initially, prior to coding the lab in assembly, I looked at the C code that was provided to us and went through each step in the code. Thereafter, I started my coding in our simulator. Since the program required us to just return the N^{th} Fibonacci number in the sequence, I created a label ‘n’ which represents which number in the sequence we want as our output. Equivalent to the provided C code, I initialized the first two numbers in the sequence (0 and 1) into two registers, r0 and r1, respectively. Then, I load the number n into a third register, r2, and then I subtract one from that number and store it into another register, which is r3. This is because, as seen in the C code, the for loop starts from n = 2 instead of n = 0. Therefore, I subtract one from the n initially and then in the subroutine for the loop, I subtract another one from the n value after going through the loop once. Thereby, accounting for n starting from 2. After subtracting the second 1 from n in our loop, I check to see if we have reached the end by checking if our n has become 0, meaning the program has finished and we can exit the loop. If we have reached 0, then we exit the loop and program can end. Before the actual loop our Fibonacci subroutine starts, we check to see if our initial n (r2) is 0 or 1, if they are, we return the corresponding number in the sequence. If not, we enter the loop, and in the actual loop I add the two initial numbers, r0 and r1, and store it into another register r4. Afterwards, because we are using the iterative approach, I move my second initial number which was r1 into the register r0. And the sum, which is in r4 is then moved into the register r1. We do this because we are iterating over this loop and to get our required Fibonacci number, we need to add the previous two numbers. Hence, every time we loop, we will be summing the latest two new numbers in the sequence. After exiting the loop, we can then move our final sum r4 into r0 as that register is supposed to be storing our return value.

1.2

This part was also about implementing a Fibonacci program, however the goal this time was to implement a recursive approach. Firstly, the input ‘n’, is loaded onto a register r0. Then we branch and link to our subroutine for the Fibonacci recursive approach. Firstly, we push all our required registers and Link Register onto our stack as specified by the callee-savee convention. The first thing to do is to check if our r0, which contains the value for ‘n’ is less than 1, i.e., is it equal to 0 or 1. If this is the case, then we can just pop the registers and our link register from the stack. This is because no action needs to be performed in our subroutine, since the register, r0, already contains the n^{th} number in the Fibonacci sequence. For n = 0, the sequence should return 0 and similarly for 1 the sequence already returns 1. However, if we that our n is not less than 1, then we go into our recursive Fibonacci operations. To implement a recursive approach, we need two add up the two previous numbers in the sequence. This implies that we need to calculate fib(n-1) + fib(n-2), this is also in the C code that we were given in the lab instructions. Therefore, first we store the value of our n into a temporary register, r1, then we load n-1 into r0. After n-1 has been stored into r0, we will call branch and link to the beginning of our Fibonacci subroutine again. Due to our r0 now being n-1, we are basically performing fib(n-1) by branching to the beginning of our subroutine. After we have calculated fib(n-1), we perform the same procedure to now calculate fib(n-2). After we have also called our subroutine on n-2, we can now add the values we have obtained for both fib(n-1) and fib(n-2). This sum is loaded onto the register r0, and then the registers that we had initially pushed in our subroutine can now be popped and we can perform branch to our link register, meaning we can exit the subroutine and end the program, as r0 now contains the n^{th} number we require from the Fibonacci sequence.

2

To implement the algorithm for this part in assembly, the first step was to read and completely understand the C code that was provided in the lab instructions. The way I completed this part, was to have the C code open in one window and code on the simulator while I kept referring to the C code. As in the C code, I also performed all the initializations as the first step in my program. Thereafter, I had to code all the nested for loops that are seen in the algorithm. The first for loop is looping through the “y” variable, therefore we first initialize our y value of 0 onto the register r0. Then we enter the body of our first for loop. As seen in the provided code, the for loop for ‘y’ runs until y < ih. Therefore, we load ih into a register r1 and compare it with r0, which is our ‘y’ variable. If we see that your ‘y’ is greater, the program has ended, and we branch to the label end. Then we enter the second for loop, and we follow similar steps as the first for loop, however, here instead of initialing our ‘y’ we work with ‘x’ instead and iw instead of iw and our compare instruction will branch to increment_y as it is the first inner loop. After that is done, we also initialize a register r8 to 0 which will represent our sum. The same steps are then followed again for the variables ‘i’ and kw in accordance with the third for loop, where our compare instructions now branches to increment_x as it is the second inner loop. Then, in our fourth loop we work with j and kh, and our compare instruction branches to increment_i. Furthermore, this is the for loop where we will do all our calculations. Therefore, we perform the additions of x+i and y+j and load the values of ksw and khw to fully perform, x+i-ksw and y+j-khw. Which represent temp1 and temp2 respectively. Then we implement the if statement in our code. We use the compare instruction again and if we see that our temp1 or temp 2 is less than 0 or greater than 9, then we exit the loop and branch to increment_j. If not, then we need to calculate the sum in this loop. We need to find the element kx[j][i] and fx[temp1][temp2]. Since our arrays for the kernel and image are set as one big array, the index for [j][i] are found using kw*j+I which is what we do with our MLA instruction, same goes for [temp1][temp2] where we use iw*temp1+temp2. So, firstly we load the address of the array which is seen when we write, ldr r3, =kx and ldr r6, =fx. Then we need to find the value at the index we require so as seen on slide 28 in the second set of lecture notes (2-isa), we use the instruction ldr r3, [r3,r2,LSL#2] and ldr r6, [r6,r2,LSL#2] respectively. Then in order to set our sum we use these values and use MLA where we store our new updated sum. After that we can finally branch to increment_j. All the increment_y, increment_x, increment_i, increment_j work very similarly where we just add 1 to our values in the registers and then branch to the corresponding immediate for loop. However, since we need to store our calculated sums in the output array, we do that in the increment_x portion of the code as visible in the C code. The storing of the sum in the array, gx, is very similar to retrieving values from the image and kernel array. However, instead of loading the values from the array into a register, we store our calculated sum, r8, into the correct index [x][y] for gx. As seen, there were several steps in this program, therefore as seen in my assembly code I used several push and pop statement and used the stack to my advantage. I was not using this approach initially. This meant I was running out of registers to use for my code, therefore after carefully going through my notes, I realized I could use the stack to fix my program and make it more efficient as well as easier to understand. Therefore, whenever I need initialize a variable, right after I push it onto the stack, and whenever I need it again, I pop it again and use it accordingly. Thus, reducing the number of registers, I required in the program and making it more efficient.

3.

This part was very similar to part 2 as we were again required to work with arrays. However, as visible through the C code, this program was considerably shorter in length. Meaning using too many registers would not be a problem, which I had encountered during the previous part. Therefore, in accordance with the C code, I first initialized all the variables that were required in the code which was the size of the array and the actual array itself. I then also created space for the output which was the same size as the initial array. Then, we load our initial values for the address of our array and our size into registers. Then we move our 'step' value into a third register, r2, which we will use to iterate over the array. Then as seen in the first for loop, we see that we iterate until we have step < size – 1. Therefore, into another register, r3, we load the value of size – 1. Then we compare in accordance with the conditions for the first for loop. If our step has become bigger than size – 1 then we branch to the end of the program. If not we initiate our 'i' variable onto a fifth register, r4 and we compare in accordance with the inner for loop seen in the C code. If we see that our 'i' variable is greater in value, then we branch to increment_step as we must exit the inner loop. If not, then we do our comparison. To make the comparison, we use the register r0 as that holds the address of the array, i.e. the address of the first element in the array. However, we need to find the actual value that is stored at that address. Therefore, using the instruction 'ldr r5, [r0,r4,LSL#2]' which is seen on slide on 28 in the second set of lectures (2-isa). Using this, the value at our i^{th} index is in the register r5. Now we need to compare this with our value at index 'i+1'. Therefore, into register 6, we load the value of i+1. Then using the same method as above, we load the value at that index into the register r7. Now we compare the two values that we have. If the value at the smaller index is smaller, then we do not need to do a swap and we can just branch to increment_i. However, if that is not the case, then using the same method we used to load the values at those index into r5 and r7, but instead of using the load instructions as we did previously, we now use a store instruction to swap the values into their new indexes. This is done by changing the value at index i+1 into r5 and the value at index i to r7. After that we are done with the for loop and we can go to the increment_i portion of the code, where we just add 1 to our value for 'i', which will compare the next two numbers in the array, which is why we branch to the second for loop after we increment 'i'. Like increment_i, we also have increment_step which just adds 1 to the register which represents our step value, after which we must go back to our first loop, hence we branch to the first for loop. And then in the end, we can make r0 point to the address of the array which will represent our output.