**Lab 1 Kristian Myzeqari 261 037 094**

The goal of the lab:

The main goal of Lab 1 was to familiarize us with using different ARM instructions and conventions. The main topics covered were compared instructions, branch instructions, shift and rotate instructions, load instructions and functions with subroutine calls. The stack's functionality was also explored using callee-save conventions during subroutine calls. The five increasingly complex tasks assigned for this lab varied from simple arithmetic routines to an image filtering routine, showing us the versatility of assembly programming language.

Task 1:

For task 1, we were assigned to write an assembly program that performs the division of two integer numbers. The result of the division, along with the remainder, must be stored in one 32-bit word. The rightmost half of the word stores the remainder of the division, and the leftmost half stores the quotient. As seen below in figure 1, to solve the problem, the program I wrote used the four argument registers where I stored the value of the dividend (A1/R0), the value of the divisor (A2, R1), the address of the result (A3, R2) and the value of the result that, initially, is set to zero (A4, R3). In addition to the registers, one subroutine named “LOOP” that performs the logical operations on the values in A1 and A2 is included in the program.

```
1  .global _start                // define entry point
2
3  // initialize memory
4  n1:        .word 11           // value of dividend
5  n2:        .word 5            // value of divisor
6  result:    .space 4           // allocated space for the result
7
8  _start:                       // execution begins here!
9          LDR R0, n1            // put the value of dividend in A1
10         LDR R1, n2            // put the value of divisor in A2
11         LDR R2, =result
12         mov r3, #0
13         BL  LOOP
14         endd: b endd
15
16 LOOP:       CMP R0, R1          // subtract divisor from dividend
17             BLE END
18             SUB R0, R0, R1
19             ADD R3, R3, #1
20             B   LOOP
21
22
23 END:        LSL R3, R3, #16
24             ADD R3, R0
25             STR R3, [R2]
26                 BX LR
```

**Figure 1: Entirety of the code for task 1**

In the subroutine, we first compare the dividend and divisor values using the CMP instruction. The following instruction reads the updated CPSR register and branches out of the loop if the dividend is smaller than the divisor. If the program remains in the loop, we can add one to R3 and subtract the value of the divisor from the dividend. Doing so every time the program follows the loop allows us to increase the quotient value by one, showing that the divisor can divide the dividend one more time.

Once the program exits the loop, a logical shift to the left of 16 bits is performed on R3 to position the quotient value to the leftmost 16 bits of the “result” word. By adding the remainder (R0) to the result register, we obtain our total value with a quotient and a remainder. Finally, the value is stored at the address of register 2 to save the final value. The program then exits the subroutine and gets finalized in an end loop.

To improve and optimize the functionality of this program, the most relevant change I could have made would be to use more complex instructions. In fact, by using an instruction such as SUBS, I could have avoided the extra step of subtracting the divisor from the dividend by removing the SUB instruction and replacing the CMP instruction with SUBS. This would have updated the CPSR and immediately subtracted the divisor from the dividend.

Task 2:

An iterative insertion sort algorithm was supplied to us for task two, which we had to use to compose our assembly program that completes the same function. The given iterative sorting algorithm uses two arguments: the number of elements in the array to sort and the array. Therefore, to write the routine, I used two argument registers for the inputs A1 for the array and A2 for the number of elements stored in the array.

After declaring my argument registers, I used a BL instruction to signal the program to enter the subroutine I wrote. But before doing so, I made sure that I respected conventions by pushing three elements to the stack: the two argument registers and the Link Register (LR). Upon entering the subroutine, I pushed into the stack all seven variable registers I used for the subroutine to respect the callee-save convention. It is essential to use the stack for variable registers to be able to reset the register values to zero upon exiting the subroutine. As seen in figure 2, after declaring the necessary variable registers, I wrote the translation of the given C code in assembly to execute the algorithm.

```
13 sortlist:
14         PUSH    {V1-V7}
15         LDR     V1, [SP, #24]   // V1 contains array address
16         MOV     V2, #1          // V2 contains i
17
18
19 FORLOOP:
20             CMP     V2, A2
21             BGE     EXIT            //
22             LDR     V3, [V1, V2, LSL #2]    // Value (array element to be sorted)
23             MOV     V4, V2          // V4 is j
24
25 WHILELOOP:
26                 CMP     V4, #0
27                 BLE     EXITWHILE
28                 SUB     V7, V4, #1
29                 LDR     V5, [V1, V7, LSL #2]            // Get value at [j-1] and load in register V5
30                 CMP     V5, V3
31                 BLE     EXITWHILE
32                 STR     V5, [V1, V4, LSL #2]
33                 SUB     V4, V4, #1
34                 B       WHILELOOP
35 EXITWHILE:
36             STR     V3, [V1, V4, LSL #2]
37             ADD     V2, V2, #1
38             B       FORLOOP
```

**Figure 2: subroutine for insertion sort algorithm**

The first step in the translation of the algorithm is to verify that the value of "i" (representing the element of the array being sorted) is less than the number of elements in the array. If i is greater than the number of elements, we can exit the subroutine. It is important to store the element to be sorted (V3) and to create a variable "j" that will iterate through the values of the array to the left of "i," comparing the stored value with the rest of the elements being checked. Before doing any sorting, a second CMP command (the first one is used for the first verification of i < number of elements) is needed to verify that the value of "j" is not negative. If "j" is negative, we exit the inner while loop and store the value in V3 into the address of the value pointed by j.

To sort all the elements, we first pick one and place it in the correct position relative to the elements to its left. So, once we have our element to sort (V3), we need to compare it to the element to its left, so as we can see on line 29 of Figure 2, we load the value at the address pointed by j-1 (on the first iteration, j = i). Next, we verify whether the element to the left is larger than the value stored in V3. If the element is greater than V3, then we shift the element at j-1 to the right by storing its value in the position to its right (line 32). This part of the loop repeats until the value to the left is smaller than the value at index "i." At that point, the value in V3 gets stored at the address of the array at index j, placing it to the right of a smaller value but to the left of a larger value. Once that step is complete, we increment "i" by one and restart the loop at line 26.

There are many ways this program could have been optimized. First, I used more registers than was necessary for the subroutine. For example, I could have omitted V1 and used the argument register to access addresses from the array. The reason why we need argument registers is to use them as arguments in a subroutine. Therefore, it was unnecessary to reassign the address to another register. Furthermore, I could have completed the program with less instructions by, once again, using CISC instructions. The more complex instructions would allow the program to execute the same functions with fewer but more complex instructions.

Task 3:

Task three is very similar to task 2, it, once again, requires us to implement an insertion sort algorithm, but in this task, it is a recursive algorithm. In assembly, what this means, is that we need to branch back to the beginning of the function, using the same arguments but possibly modified. To efficiently accomplish this task, we were first required to write the C program associated with this task (figure 3). The following code encapsulates the same logic as the C code provided for task 2 but is slightly modified/

```
1   void InsertionSort(int i, int arr[], int n)
2   {
3       if (i < size)
4       {
5           int j;
6           int value = arr[i];
7
8           for (j = i; j > 0 && arr[j-1] > value; j--)
9               arr[j] = arr[j-1];
10          arr[j] = value;
11
12          InsertionSort(++i, arr, n);
13      }
14  }
```

**Figure 3: recursive insertion sort in C**

The logic for task 3 is essentially the same as for task 2, except that there is only one loop in the recursive program. We substitute the outer for-loop with the recursive call at line 12 and the inner while loop with a for-loop. To ensure that the sorting mechanism goes through all the elements of the array, it is essential to increment "i" in the recursive function call. This will allow us to visit each element of the array. In assembly code, the recursive call entails pushing the LR with every single call. The primary difficulty in this task was keeping track of all the elements pushed into the stack and popping them at the appropriate time. As seen in figure 4, I pushed and popped the LR before and after the recursive call.

```
36 exitfor:
37     STR     V3, [A1, V1, LSL #2]
38     ADD     A3, A3, #1
39     POP     {V1-V4}
40     PUSH    {LR}
41     BL      sortlist
42     POP     {LR}
43
44 exit:
45     BX LR
```

**Figure 4: push/pop of LR and exit branch**

When the array is sorted (verified by checking if the value of i is greater or equal to the number of elements in the array, the program counter is set to the "exit" branch. As previously mentioned, the LR is pushed before every recursive call, so the stack contains N (number of elements) LR values. The "BX LR" at line 45 indicates that the program shall branch to the saved value of the LR, which is at line 42. Since the next instruction is, once again, BX LR, the program branches back to line 42, where the process restarts, oscillating between the "POP" at line 42 and the "BX" at line 45 until the LR points back to the beginning of the program where the "sortlist" subroutine is initially called, as such, every single LR value gets popped, and the program continues as it should.

The optimizations for this routine are very similar to the ones mentioned for tasks 1 and 2. There would be fewer instructions if I considered using CISC instructions and taking advantage of their efficient functionalities.

Task 4:

The general structure of task 4 is the same as task 3, except that the recursive call inside the subroutine should not be a BL instruction but a B instruction. No registers should be pushed in the stack either for this task. To complete this section of the lab, I used the same code that I had for task 3 but substituted the BL instruction from line 41 of task 3 with a B instruction. I also removed the push and pop instructions before and after the recursive call. Moreover, I had to remove all the push instructions I used for the V- registers.

Since the code is copied from task 3, the optimizations are the same as the ones for task 3.

Task 5:

Task 5, the most challenging task of the lab, consisted of applying a median filter to perform noise reduction on an array of 100 pixels. To solve this task, I had to implement a program that took the median values of four different channels (RGBA) of 25 pixels to create a smaller image. Out of the many challenges I found in this task, the toughest one was to figure out how to make the "window" selecting the pixels being filtered move from one section of the array to the next. I did not manage to complete this portion of the lab successfully. Still, with many hours spent on this task, I separated the channel values from 25 pixels into four different arrays (one for each channel).

To do so, I used the STRB and ROR instructions to store the channels in their respective arrays, one byte at a time (figure 5).

```
62              STRB V4, [V5, V1]       //store alpha channel
63              ROR V4, V4, #8
64
65              STRB V4, [V6, V1]       //store blue channel
66              ROR V4, V4, #8
67
68              STRB V4, [V7, V1]       //store green channel
69              ROR V4, V4, #8
70
71              STRB V4, [V8, V1]       //store red channel
72
73              ROR V4, V4, #8          //Return to original state
74              ADD V2, V2, #1          //Increment j
75              ADD V3, V3, #4          //Increment element being stored by 4 to get next byte in memory
76              ADD V1, V1, #1
77              b forloopJ
```

**Figure 5: STRB and ROR instructions demonstrated**

The snippet of code above shows how I went through one byte at a time and incremented the value of my single counter variable (j) to keep track of how many pixels I visited for the arrays. I ensured that once a row of five pixels was complete, I would move on to the row below instead of continuing with the next element in the array of pixels. This ensured that I always had the suitable pixels entered in the array for sorting.

To sort the array, I used the recursive insertion sort program from task 3, adapted to sort the information one byte at a time instead of one word at a time. Like that, I always sorted the correct number of information and kept the bytes independent (each byte on its own). After sorting the arrays, I selected the median value for each of them and put back all of the channel values in their respective byte positions in the word representing a single pixel (figure 6).

```
222 getmedian:
223     PUSH    {V1-V8}
224     LDR     V1, =array_space_red
225     LDR     V2, =array_space_green
226     LDR     V3, =array_space_blue
227     LDR     V4, =array_space_alpha
228
229     LDRB    V5, [V1, #13]
230     ROR     V5, V5, #8
231     LDRB    V6, [V2, #13]
232     ROR     V6, V6, #16
233     LDRB    V7, [V3, #13]
234     ROR     V7, V7, #24
235     LDRB    V8, [V4, #13]
236     ADD     V5, V5, V6
237     ADD     V5, V5, V7
238     ADD     V5, V5, V8
239     STR     V5, [A1, #0]
```

**Figure 6: snippet of code showing how all the elements are placed into their respective slots in a word**

To save the value of the word created, I stored the result in the memory address allocated to the array containing the result of the computations.

The most important change I would have liked to make in my program would be the number of function calls that I used. When sorting the arrays, I did not try to implement any optimization techniques, in fact, I created a subroutine for each array, and sorted the arrays independently (figure 7). Evidently, this makes my program much slower than it could be, and costed me more time than if I would have looked for way to sort the arrays in the same subroutine.

```
31 _start:
32     LDR A1, =result
33     LDR A2, =input_image
34     LDR A3, length
35     MOV A4, #1
36     PUSH    {LR}
37     BL      organizeArrays
38     BL      sortArrayAlpha
39     BL      sortArrayBlue
40     BL      sortArrayGreen
41     BL      sortArrayRed
42     BL      getmedian
43     POP     {LR}
44 end: b end
45
```

**Figure 7: all subroutine calls used in the program.**

Once again, I could have also optimized my program by making use of CISC instructions to reduce the number of instructions in my program.

**Collaborations:**

Nabil Amimer: Nabil helped me with task 5 by giving me extra explanations for the problem, he also guided me at the beginning of task 5 by giving me tips to write the correct code.

Matthew Cabral: Matthew helped me to debug task 2, I was having issues with the stack and he explained to me how to make sure that all the push and pop instructions were placed in the right spots.