Laboratory Report 1

Due February 13th, 2023

Marc Bretones - 260973002

1.
a. Matrix Multiplication

Introduction:

Matrix multiplication is an essential operation in various fields, including computer science, physics, and engineering. In this section, I discuss the successful execution of matrix multiplication with a size of 2, which also works with matrices of size 3. The execution is done with registers R0-R3 for Matrices a, b, c & size of the matrix respectively.

Method:

Three loops are used to perform the multiplication. More specifically, for loops are implemented in ArmV7, each with an internal tracker (“i” in higher level languages) - "rowloop," "colloop," and "iterloop," corresponding R4, R6, and R8. The function “mm” is called with BL in “start”, and nested loops succeed one another such that the C twin is properly executed.

Each loop uses “CMP” to compare the size of the matrix with the former’s internal counter. Before incrementing the counter’s value into registers, the “BGE” instruction sends information such that I branch out of the loop or keep iterating depending on the current value.

“MUL”, “ADD” and “STRH” are instructions used in the intermediary loop to compute “*(c + row*size + col) * 2= 0”. The value is “STRH” (stored) in R12’s address. Further optimization of the code would have entailed using MLA for multiplication and addition combined.

The addresses of “*(a + row*size + iter) X *(b + iter*size + col)” was computed in a similar fashion in the furthest inner loop and stored in R10. These values was then loaded to and added to the previous value in R5 “*(c + row*size + col)” which started at 0. Lastly, the index of “c” gets shifted by accessing its address and changing it with a multiple of 2 (half word), to start the process over for other values of the matrix.

Conclusion, Challenges, Possible Optimization:

In conclusion, the successful execution of matrix multiplication requires careful planning and effective use of registers and loops. By disintegrating the function into several components and utilizing MUL and ADD, along with LDRSH and STRH instructions, I was able to efficiently compute matrix multiplication with a size of 2 and 3, the latter provided 10 zeroes are assigned to Matrix “c”.

Important challenges in this first exercise was to learn ARM assembly, a first for me. Further, the concept of loops was non-trivial to implement as I kept having issues with counters, which must be strategically placed in order to not end in an infinite looping situation.

As for improvements of this function, many roads can be taken: mainly the reuse of registers by a more efficient use of PUSHing and POPing registers. Recomputing values is a major hurdle my code encounters to be more efficient. Making better use of Registers such that registers aren’t overwritten would be a solution. Further, the Winograd algorithm would reduce calculations & improve the algorithm.

1.

b. Winograd Matrix Multiplication

Introduction:

The Winograd algorithm is a fast and efficient method for matrix multiplication. In this section, I discuss the successful implementation of the Winograd algorithm for multiplying two matrices of size 2x2. Specifically, I will detail the steps involved in computing the values of the resulting matrix, which is denoted by C.

Method:

I begin by computing the following intermediate values and storing them into their respective registers: “u = (cc - aa)(CC - DD)” into R10, “v = (cc + dd)(CC - AA)” into R11, “w = aaAA + (cc + dd - aa)*(AA + DD - CC)” into R12. By using the instructions “ADD”, “MUL”, “MLA”, “SUBS” and LDRSH (the latter to load the values into given registers). Here, aa, bb, cc, and dd denote the elements of the first matrix A, while AA, BB, CC, and DD denote the elements of the second matrix B. aaAA denotes the product of aa and AA.

Next, I compute the elements of the resulting matrix C as follows: “c = aaAA + bb*BB”, “(c + 02 + 1) = w + v + (aa + bb - cc - dd)*DD”, “(c + 12 + 0) = w + u + dd*(BB + CC - AA - DD)” and store the value into R4 with “STRH” in a similar fashion, by accessing the indices of matrix “c” in increments of 2 . Please note that the notation (c + 02 + 1) refers to the element of the resulting matrix C that is in the 0th row and 2nd column plus 1. Similarly, (c + 12 + 0) refers to the element of C that is in the 1st row and 2nd column.

Conclusion, Challenges, Possible Optimization:

In conclusion, the Winograd algorithm provides a fast and efficient method for matrix multiplication. By breaking the computation down into several intermediate steps, I can compute the elements of the resulting matrix in a way that minimizes the number of calculations required.

One challenge in implementing this procedure is keeping track of the various intermediate values and their corresponding indices. Careful attention must be paid to the indexing in order to ensure that the correct elements are computed and stored in the resulting matrix.

Possible optimizations for this procedure include using parallel computing techniques to speed up the computation of the intermediate values, as well as optimizing the memory usage by minimizing the number of temporary variables used in the calculation, that is - making more efficient use of registers to store values and reuse them more simply, by making use of PUSHing and POPing more efficiently.

2.

a. Binary Search

Introduction:

The binary search algorithm is a commonly used technique for locating the position of an element in a sorted list. In this section, I discuss the successful implementation of the binary search algorithm in ARM assembly language. Specifically, I will detail the instructions used, the registers employed, and the loop structure of the algorithm.

Method:

The algorithm begins at the label "_start", where the starting address of the array is loaded into register R12. I then load the values 4, 0, and 7 into registers R1, R2, and R3, respectively, to serve as the value to be searched and the low and high indices of the array. A branch-and-link instruction (BL) is then used to call the binary search subroutine.

The subroutine, "binarySearch", employs a stack frame to store and restore the contents of registers R4-R11. The while loop structure is implemented using a label "while" and a branch-and-execute (BX) instruction. The loop consists of two branches labeled "low_GE_high" and "low_LT_high", which together implement a binary search in the array.

The first branch, "low_GE_high", compares the values of the low and high indices and branches to the second branch if the low index is less than the high index. If the value being searched equals the element at the high index, the value is immediately returned; if not, the value being searched is compared to the element at the low index. If they are equal, the index of the value is returned; if not, the algorithm returns -1.

If the low index is less than the high index, the second branch, "low_LT_high", is executed. The middle index of the array is computed as the sum of the low and high indices divided by two, and is stored in register R5. If the value being searched equals the element at the middle index, the index of the value is immediately returned; if not, the value being searched is compared to the element at the middle index. If the value is greater than the element at the middle index, the low index is incremented to the middle index plus one and the loop is re-executed. If the value is less than the element at the middle index, the high index is decremented to the middle index minus one and the loop is re-executed.

Conclusion, Challenges, and Possible Optimizations:

In conclusion, the binary search algorithm is an efficient and commonly used technique for locating the position of an element in a sorted list. In the implementation discussed above, I used a while loop structure comprising two branches to achieve this goal.

One challenge in implementing this algorithm is keeping track of the low and high indices and the value being searched. Careful attention must be paid to the branching instructions and the use of registers to ensure that the correct elements are being compared and that the correct index of the value being searched is returned.

Possible optimizations for this algorithm include using the recursive binary search algorithm, parallel computing techniques to speed up the search and optimizing the use of registers to minimize the number of temporary variables used in the computation. By carefully designing the algorithm and the use of registers, the speed and efficiency of the binary search can be improved.

2.

a. Binary Search Recursive

Introduction:

The binary search algorithm is a popular search algorithm that operates by dividing a sorted array in half, and repeatedly halving the search area until the target value is located. In this section, I will discuss a recursive implementation of the binary search algorithm in assembly language, along with an explanation of the algorithm's logic and functionality.

Method:

The binary search algorithm begins by initializing the starting index R2 and the ending index R3 of the search range. These are set to 0 and 7, respectively. The target value to be searched is set to R1. The array to be searched is loaded into the register R12. The BL command is then called to invoke the binarySearch subroutine.

The binarySearch subroutine checks if the starting index is less than the ending index. If so, the program continues with the x_EQ_array_lowidx condition, in which the element at the ending index is compared to the target value. If they match, the index is stored in R10, and the subroutine ends. If not, the x_NE_array_lowidx condition is executed, in which R10 is set to -1 to indicate that the target value was not found. The subroutine then ends.

If the starting index is greater than or equal to the ending index, the program proceeds with the low_LT_high condition, which calculates the midpoint of the search range. The program then executes the x_EQ_array_mid condition, which compares the midpoint element to the target value. If they match, the index is stored in R10, and the subroutine ends. If not, the program continues with the x_NE_array_mid condition. If the target value is less than the midpoint value, the x_LT_array_mid condition is executed, and the binarySearch subroutine is recursively called with the ending index set to the midpoint - 1. If the target value is greater than the midpoint value, the x_GT_array_mid condition is executed, and the binarySearch subroutine is recursively called with the starting index set to the midpoint + 1.

Conclusion and Challenges:

In conclusion, the recursive binary search algorithm is an effective and efficient means of searching a sorted array for a target value. By repeatedly dividing the search area in half, the algorithm quickly reduces the search area until the target value is located. The implementation of the algorithm in assembly language requires careful attention to detail and a clear understanding of the algorithm's logic.

One challenge in implementing the recursive binary search algorithm is the potential for stack overflow when dealing with a large array. This can be addressed through optimization of the code, such as minimizing the number of registers used and using tail recursion to reduce the number of function calls. Another challenge is the possibility of an infinite loop, which can be prevented by proper initialization of the search range and careful handling of boundary cases.