Sofia Velasquez Sierra
260967314
February 10th 2023

LAB 1: Report

1.

The first part of the Lab consists of implementing two matrix multiplication codes in assembly language. The first code uses *for* loops, and the second operates on the winograd matrix multiplication technique.

A. Matrix multiplication

Total program size: 232 bytes
Executed instructions: 260
Data loads: 36
Data stores: 20
Total memory accesses: 316

Approach:

The first code was approached by following the C code 'line by line'. First, A1, A2 and A3 passed the addresses of the matrices a, b and c respectively. A4 passes the length of one side of the arrays, *size*, when using a 2x2 matrix, it passes the value 2. My initial approach was to implement three subroutines; one for each *for* loop, but I had to add an additional one (*mm*) to avoid initialising the *row* of the matrices at each iteration.

The first subroutine (*mm*) initialises the *row*, by moving 0 into V1, iterating through the rows of the matrix and either branching to *mmDone*, ending the code, or setting V3 to 0 (the column counter) and entering the following loop. The subsequent subroutine *mmIter* iterates through each column of the matrix, incrementing the value of V1 (*row*). It will either exit the loop, incrementing the row and branching back to *LOOP1,* if *col* is greater or equal to *size*, or execute the computations that follow (*(c + row * size + col) = 0) by using the MUL, ADD, LSL instructions and storing the result in V6. In the latter case, the loop terminates by initialising the *iter* counter to 0 (setting V5 to 0). The last subroutine (*mmLoop*), compares *iter* to *size,* incrementing *row* and branching to *mmIt*er if *iter* is smaller than *size,* or remaining in the current loop. *mmLoop* will then use the instructions MUL, ADD and LSL to to perform the subsequent computations (*(a + row*size + iter) * *(b + iter*size + col)) on the given *row* and *col* addresses of a and b, storing the final product at the given *col* and *row* of matrix c and finally branching to *mmLoop*. Finally, when *LOOP1* branches *mmDone*, the used registers (V1-V8) are popped and the instruction BX LR is called, ending the code and entering the infinite loop.

Challenges:

This was my first time coding in assembly and the first exercise of the lab so I encountered a few challenges. First of all, understanding how to use different subroutines and the branching between them was slightly confusing. Also, using registers and understanding how the memory accesses exactly worked needed some getting used to especially when it came to the iterations of a subroutine. I found that the emulator was a rather straightforward but I encountered some issues when it came to the debugging of the code and understanding why I was encountering errors by looking at the memory.

Shortcomings:

This code is not very efficient; it is somewhat long for what it could be (there are 230 memory accesses). Going back and forth between subroutines reduces the efficiency and

optimization of the code. I could also have optimised my code by using three subroutines instead of four which made my code slightly redundant.

B. Winograd Matrix Multiplication

Total program size: 264 bytes
Executed instructions: 55
Data loads: 26
Data stores: 12
Total memory accesses: 93

Approach:

The second part of matrix multiplication uses the winograd method. The code starts by passing the addresses of the matrices a, b and c into A1, A2 and A3 respectively, the calls *wmm22*, the main function. After pushing registers V1 through V8, the computations begin. To compute *u*, I loaded cc, aa, CC and DD into registers V1-V4, then used the instructions SUB and MUL to find *u,* which at this point is located in register V5. To compute *v*, I loaded dd and AA into V6 and V7, and used the ADD, SUB and MUL to find *v*. The latter instructions' addresses were put in registers that had previously been used, so overwriting their previous values in order to perform the current necessary loads and computations. Next, I computed w in the same manner; overwriting previously used registers to load and perform the current computations.

To compute the element at each position of the *c* matrix, I followed this method of overwriting registers to compute the current needed operations making sure to not overwrite the registers containing *u, v* and *w* (V5, V8, V2). The computed values were each stored in their respective row and column. The function ends by popping the used registers V1-V8, and entering the infinite loop.

Challenges:

For the winograd part of this lab, the main challenge I encountered was using registers. I had first thought about loading all the values from matrices *a* and *b,* but instantly realised I would not have enough registers to pursue this approach. Then, I tried to use PUSH and POP, with different values such as *w, u, v,* to free up registers while the *pushed* values were not in use. However, since this was my first time using the stack I quickly ran into issues I did not know how to solve at first. Finally, I decided to maintain a more simpler approach, although less optimised and longer, which was to overwrite registers and even load the same value several times throughout the code.

Shortcomings:

Other shortcomings could appear if we tried to use matrices other than 2x2 matrices, since the register overwriting is very long and complex and also the code is written to store a 16-bit number, which could not be the case for every matrix size. Also,the code could have been more efficient if I had used the stack.

I. Performance analysis

Comparing both codes, we observe that the winograd has significantly less memory accesses than the looping code (93 vs 316). However, it is the opposite for the program size (264 vs 232). The differences in memory accesses can be explained due to the amount of

computations have to be redone due to the looping, since it has to iterate through every column, row and and each element of every matrix. As opposed to the winograd, who maintains a significantly low number of memory accesses, because there are no counters, comparisons between values and branching between loops. Overall, despite my approach to winograd which was not the most efficient, this code remains less complex than the first one.

2.

The second part of the lab implements a binary search algorithm in an iterative and recursive way.

A. Iterative Binary Search

Total program size: 184 bytes
Executed instructions: 44
Data loads: 15
Data stores: 8
Total memory accesses: 67

Approach:

For iterative binary search my approach was to follow the C code as it was coded, so to implement a for loop. The function uses the address of the array, an integer *x*, a low index and a high index, whose addresses were passed to A1, A2, A3 and A4 respectively. The function *binarysearch* is then called.

The code begins by pushing registers V1-V8, and moves into the first subroutine *while*. Here, I compare A3 to A4, and either continue onto the following subroutine or branch to *mid* if the low index is smaller than the high index. In the case were the code continues to the following subroutine, it enters *if;* here, the low index of the matrix is loaded to V2 and compared to *x.* If these two values are equivalent, we branch to *return*, else -1 is returned. If the subroutine *return* is entered, the current value of *lowidx* is entered.

When the subroutine *mid* is entered, the **mid** point of the array is defined (V1) and we branch to the following subroutine *early*. Now, the middle value of the array is loaded into V3 and we compare it to *x.* If the values are equal the function branches to *returnmid*, else it increases *lowidx* by one and decreases *highidx* by one, and branches back to *early.* When *returnmid* is entered the current value if *mid* is returned.

Finally, V1-V8 are popped and we enter the infinity loop.

Challenges:

I found that this part of the Lab did not present any significant challenges, it was straightforward and I had already understood how to organise the way registers are used.

Shortcomings:

The code could be optimised by avoiding to use extra subroutines when it is not needed. I added subroutines to my code and branched to the subsequent subroutine at the end of each one of them, which was redundant but at first helped with my understanding of the code in assembly. Also, I repeated the instruction CMP between A2 and V3 in the same subroutine, which could have been avoided.

B. Recursive Binary Search

Total program size: 192 bytes

Executed instructions: 37
Data loads: 13
Data stores: 7
Total memory accesses: 57

Approach:

For this part, the binary search algorithm is implemented in a recursive way. It is somewhat similar to the previous part and so I based myself of off my previous code. The code begins by assigning the address of the array, an integer *x*, a low index and a high index to A1, A2, A3 and A4 respectively, and then branches to the function *binarysearch* using BL, which sets the LR to the next instruction.

Entering *binarysearch*, the registers V1-V3 are pushed, then A3 and A4 are compared, and either the function continues onto the following subroutine or branch to *mid* if the low index is smaller than the high index. In the case were the code continues to the following subroutine, it enters *if;* here, the low index of the matrix is loaded to V2 and compared to *x.* If these two values are equivalent, we branch to *return*, else -1 is returned.

When the subroutine *mid* is entered, the **mid** point of the array is defined (V1) and we branch to the following subroutine *early*. Now, the middle value of the array is loaded into V3 and we compare it to *x.* If the values are equivalent, the function branches to *returnmid*. Otherwise, *x* is compared to the value at position *mid*. If *x* is greater, the value of *lowidx* is updated to *mid+1*, else the value of *highidx* is updated to *mid-1*. It continues by popping V1-V3, pushing LR and branching back to the initial function using BL. After this branching, the link register is popped and BX LR is called.

If the function branches to *returnmid, mid* is returned. If it branches to *return, lowidx* is returned. The code ends by popping V1-V3, calling BX LR and entering the infinity loop.

Challenges:

The main challenge encountered in this part was understanding where to place the POP and PUSH instructions of both the registers and the link register. At first, this was causing many errors and breaking the code.

Shortcomings:

During the demo of this particular code, it was not outputting -1 when *x* was greater than the highest index. This is something that should be optimised.

I. Performance analysis

Comparing both codes we observe that the recursive binary search has a larger program size than the iterative (192 vs 184). As opposed to the number of memory accesses, which is greater for the iterative binary search (57 vs 67). The iterative binary search should overall have smaller values, which means that my code should have been optimised. The loop in the iterative bs should execute less instructions, as opposed to the recursive whose main function needs to be recalled several times.