ECSE 324 – Computer Organization (Fall 2018)

Lab 1 – Introduction to ARM Programming

Group 12 – Alain Daccache & Michel Abdel Nour

In this lab, we worked with the DE1-SoC Computer System and programmed common routines in ARM assembly. The system's architecture consists of an ARM Cortex-A9 processor and peripheral components of the FPGA on the DE1-SoC board. We used the Intel FPGA Monitor Program 16.1 as our IDE.

Programming the Computer System in ARM assembly:

The first program consisted of finding the maximum number from a list of 'NUMBERS' with length 'N'. After writing it in the editor, we saved it in a file 'part1.s' and added it as a source file in a new project 'G12_Lab1'. After compiling it and fixed some syntax errors, then compiling again, we loaded it on the board, and dynamically observed the different registers along with their memory values on the IDE.

After running it and pausing it, we can find the correct maximum number of the list (which is 8) in R4 which points to the result location. After going to the "Memory" tab, we notice that the value stored at address 0x00000038 is 00000008 (the max of the list)

Studying the code helped us get familiar with the working environment and understand the common operations such as LDR, CMP, SUBS, BEQ, etc.

Calculating the Standard Deviation:

For this part, we had to find the maximum and minimum number of the list, then subtract the minimum from the maximum and divide by 4. This is a simpler way of calculating the standard deviation without requiring square root operations.

Approach taken:

1. We started by storing the first number of the list into the registers that should hold the maximum and minimum numbers. We assume that is the current max and min of the list
2. Find the maximum was easy, we used the max loop from the previous part. After decrementing the counter, we loaded the next number, stored it in another register, then compared it to the current maximum. If the current maximum is more than the next number, we branch back to the start of the loop. Else, the latter becomes the current maximum, and we branch back to the start of the loop
3. Then, we made sure to reset the counter to the number of elements, and the pointer to point to the address of the first element.
4. Finding the minimum was similar to finding the maximum. We do the same thing at first, then we compare the next number to the current minimum. If the current minimum if greater than the next number, we update the register storing the current minimum to store this next number, then we branch back to the loop. Else, we directly branch back to the start of the loop
5. Then, we subtracted the minimum from the maximum and stored the result in a register with the SUB command. To divide the result by 4, we notice that $4 = 2^2$ so dividing by 4 is equivalent to shifting right by 2 bits, so we used the LSR command to perform the instruction.

Improvements: We could have made this algorithm more efficient by searching for both the maximum and the minimum in the same loop. But to simplify the code to give each block its purpose, we split the task in two. This might be useful for refactoring purposes in future labs.

Challenges: This part was straightforward, as we refactored the code from part 1 for maximum, and slightly modified the CMP instruction and consequences for the minimum. It did take us time to understand how the CMP command works and which flag(s) it raises. We also accidently used arithmetic shift at first and we did not understand what we were doing wrong.

Centering a list of numbers:

Approach: To center a list of numbers, we have to subtract the average of the list from each number and storing it in place. To find the average, we have to calculate the current sum and divide it by the number of elements. Assuming the number of elements is a power of two, performing this step requires logical right shift. To find out by how much to shift, we find the logarithm of the number of elements. To do so, we keep dividing the number of elements by 2 until we it reaches 1, while incrementing a counter (representing the logarithm), in each iteration.

Challenge: The code was a little bit longer than the other exercises, but we really wanted to divide and conquer in this exercise, and dedicated a portion of the code to each sub-task. Challenge-wise, figuring out by how many bits should we shift to divide took us some time, but then we realized we can implement our version of a logarithm base 2.

Sorting a list of numbers (Bubble Sort)

Description: The bubble sort makes multiple passes through a list. It compares adjacent items and exchanges those that are not in order. Each pass through the list places the next largest value in its proper place, starting from the maximum. This passing procedure is repeated until no swaps are required, indicating that the list is sorted.

Challenges:

- We got confused on how to implement a for loop that starts at the second element and ends at the last i.e. comparing current element to previous one, stopping at the last element. We figured out that we can start at the first element and end at the before-to-last element i.e. comparing current element to next one, stopping at the before-to-last one. This is to avoid running the loop an extra iteration and accessing an area in memory that is not related to the list of numbers.
- Also, when swapping the elements in the swap branch, we forgot to empty the used variables.
- Finally, figuring out how to implement a Boolean value took us sometime until we noticed we can just use immediate values #0 and #1 to indicate false and true, respectively.
- Otherwise, translating a given pseudocode into assembly was easy. It was interesting to understand how a for loop inside a for loop works in assembly.

What we learned:

We learnt to code block by block, debugging every time, instead of writing dozens of line then compiling and getting errors in which we don't understand their cause. We became more familiar with the debugging process. We put breakpoints to reach areas of code of interest, or to skip parts of the code that we know work. Then we jump to instructions one by one, keeping track of the how the values that are stored in the register change, and the content stored in memory addresses change. When we see a discrepancy between what we expected and what the result actually is, we pinpointed where is the problem.

When we want to modify the contents of a variable (i.e. RESULT), we first load the address of result in a register A in order to point to that location. Then, we use ANOTHER register B that stores the result (i.e. storing the current sum). In fact, an error we made is to add the next number of the list to the current sum in the register that points to the result location, therefore updating the address with the numbers of the list! Therefore, after we get the result we want, we store the contents of the register B into A (de-referenced)

When we branch to a loop, we made an error where we reset the decremented N (counter) as the first instruction in the loop, therefore always resetting it and never ending the loop. Other problems had similar causes i.e. using the counter in a loop then in another loop without resetting it. Therefore, we learnt that it is good practice to add an intermediary branch in between branches where we can store the results we want in the corresponding registers as well as reset the counter and pointer.