Lab1 – Arrays and Functions

ECSE-324 Computer Organization

Sen Wang (260923645)

Friday 16, October 2020

McGill University
Montreal Quebec, Canada

Part 1_1: Square root of an integer using the SGD technique (iterative solution)

Brief Description:

This assembly program takes an integer as an input and calculates its square root using the SGD technique. For our program, we used 100 iterations. The code can be separated into 6 main parts. The first part of the program is variable declaration. Here, it used MOV to store values/constants into their respective registers. The second part is the main loop where the program computes the step variable and compares step to t. The third part is the GreaterThan branch which assign step variable to t if step is greater than t. The fourth part of the program is LesserThan branch which assign step variable to -t if step is lesser than -t. It branches to GreaterThan or LesserThan using CMP inside the loop. The fifth part of the program decrements xi by step at each loop iteration and decrements the counter for the loop. Finally, the last part is the Stop branch which calls itself recursively. It branches to Stop when we are outside of the loop.

Approach Taken

As described in the brief description, the approach taken for this program is the use of Branch Instructions (subroutines) and conditional execution. For example, in the main loop, I used CMP to compare values. If step is greater than t, then the program branches to GreaterThan using BGT. Similarly, if step is smaller than -t, the program branches to LesserThan using BLT. To implement the loop logic, the program uses BGT to loop after we decrement the loop counter. More precisely, the loop counter starts and 100 and decrements by 1 at each iteration. If the counter is greater than 0, then the program branches to Loop.

Challenges Faced and Solutions

Since the first part of the lab is relatively simple compared to the other parts, no particular challenges faces. The only difficulty I faced for this first part of the lab would be the use of conditional execution. At first, I couldn't really figure out how the comparison worked. The issue was quickly solved after I revised the notes and after some internet searches.

Possible Improvement to the Program

This program works but several improvements to the program could be made. For example, this program used a lot of registers. Some registers are only used as temporary variables to store intermediate calculation values. Reusing some registers can then reduce the total number of registers. Another improvement would be the use of less branches. In this program, I used an additional branch to just decrement the values.

Part 1_2: Square root of an integer using the SGD technique (recursive solution)

Brief Description

This assembly program takes an integer as an input and calculates its square root using the recursive SGD technique. This program recursively calls the function branch 100 times. The program can be separated into 3 main parts. The first part of the program is variable declaration. Here, it used MOV to store values/constants into their respective registers. The second part is the main function. Here, we compute the step variable and compares it to t. We used MOVLT and MOVGT for conditional execution. Finally, the last part is the End branch which also calls itself recursively. It branches to End after calling Function 100 times.

Approach Taken

Just like the first part, this program also uses branch instructions (subroutines) and conditional execution. But here, I simplified the program by using conditional instructions like MOVLT and MOVGT. Therefore, I eliminated the use for more branches. Otherwise, the program calls itself recursively 100 times to execute the computation.

Challenged Faced and Solutions

Just like part1_1, no particular challenges were faced. The only bug or compilation issue that I had was the Function Nesting too Deep error. After some digging in Piazza, I found out that this is a feature that checks is the function call nesting depth exceed 32. In fact, normally when a function call that goes beyond that often result to an error. However, for our case, we wanted at least 100 nested function calls. So, all I had to do was to disable this feature in the program settings.

Possible Improvement to the Program

For part1_2, the program works but several improvements to it could be made. Just like part1_1, this program used a lot of registers. Even though it used a lot less than part1_1 due to the fact I used more conditional instructions like MOVLT and MOVRT, a lot of registers are still being used just as temporary variables to store intermediate calculation values. Reducing the number of registers used could make the program more flexible in the future if we want to make modifications on it.

Part2: Norm of a Vector Array

Brief Description

This assembly program takes an array as an input and calculates its norm. To simplify the calculations, we only consider the length of the array to be a power of 2, thus the division can be implemented by a right-shift operation. This program consists of 6 main parts. The first part is variable declaration where we use R0 to store the start address of the array. The second part is the WHILE branch which calculates log2_n using LSL. The third part is the main loop branch where we add the square of each element of the array together. The fourth part is the branch where the program uses the previous calculated sum and uses ASR by log2_n to divide by the length of the array. The fifth part of the program is SQUARE_ROOT_FUNCTION branch where I reused code from part1_2 to compute the square root. Finally, we have an end branch to stop the execution of the program.

Approach Taken

This program uses multiple branch instructions (subroutines) to separate the core logics of the code. In order to use less registers in the program, I decided to use the MLA instruction to calculate the sum of the square of each element of the array. Additionally, to iterate through the array, I used the post-index mode (e.g LDR R7, [R0], #4). Other than that, since the square_root_function branch is the same as part1_2, multiple conditional executions were used.

Challenges Faced and Solutions

The biggest challenge I faced for this part of the lab would be the use of registers. The parts before square_root_function was already using a lot of registers. However, I did not know I had to reset them so the outputs I was getting at first were wrong. I solved the problem by going through the code line by line using the debugger and verifying if the registers values were correct.

Possible Improvements to the Program

Several improvements could be made to this program. Since the square_root_function branch is the same as part1_2, we can also assume that a lot of registers are just being used to store temporary intermediate calculation variables. The disadvantage of this flaw is especially shown in this part of the lab because I had to reset the registers before computing the square root.

Part3 Centers a Vector Array

Brief Description

This assembly program takes an array as an input and centers it. Just like part2, to simplify the calculations, we only consider the length of the array to be a power of 2, thus the division can be implemented by a right-shift operation. This program also consists of 6 main parts. The first part is multiple variables declarations where we use R0 to store the start address of the array. The second part is the WHILE branch which calculates log2_n using LSL. The third part is the main Loop branch where we add up all the values in the array in order to calculate the mean in the fourth part. In the fourth Mean branch, the program computes the mean by computing an ASR by log2_n calculated previously. In the fifth Center branch, the program subtracts the average from every sample of the array. Finally, the program branches to End which stop the execution of the program.

Approach Taken

Just like part2 of the lab, this program also uses multiple branch instructions (subroutines) to separate the core logics of the code. Additionally, to iterate through the array, I used the post-index mode (e.g LDR R7, [R0], #4). The rest of the program was very standard and did not require any special instructions since the logic were simple.

Challenges Faced and Solutions

For this part of the lab, I did not face any particular challenge. In fact, most the difficulties for part3 were already being solved in previous parts (use of registers, conditional execution, etc).
The only point of confusion I faced during the code implementation was changing the values of the array and load them back to the memory. However, this was quickly fixed by revisiting the notes with some internet searches.

Possible Improvements to the Program

Several improvements could be made to this program. Just like other parts of the lab, the main improvement would be the use of less registers since they are only being used to store temporary intermediate calculation variables.

Part4 Selection Sort Algorithm

Brief description

This assembly program takes an array as input and performs the selection sort algorithm on it. This program consists of 6 main parts. The first part is multiple variables declarations where we use R0 to store the start address of the array. The second part of the program is the first_loop branch where we set the current value of the array as tmp variable and the cur_min_idx as i. We then have a second nested loop (second_loop branch) which is the third man part of the program. Here we compare tmp to values of the array and branches to the Update branch (fourth main part of the program) and update tmp and cur_min_idx if necessary. After looping through the second_loop, the program branches to swap branch which is the fifth main part of the program. Here, we swap the value of tmp with the value at cur_min_idx of the array. Finally, the program branches to End which stop the execution of the program.

Approach Taken

Just like other parts of the lab, this program also uses multiple branch instructions (subroutines) to separate the core logics of the code. Additionally, in order to load the values of the array into the tmp variable, I used “LDR R3, [R0, R2, lsl#2]. Here, R3 is where the variable tmp is stored. R0 is the start address of the array. R2 is the counter to iterate through the array. Finally, I used lsl#2 because an integer uses 4 bytes in memory. The same approach is taken for the second loop. Finally, in order to swap the values, similar instructions were used. Additionally, I had to use a tmp variable in between the swap in order to properly store the values.

Challenges Faced and Solutions

The main challenge I faced for this part of the lab is the swapping part of the algorithm. More specifically, the line of code where we store the value at cur_min_idx to the address of the tmp variable caused most of the issues because I had a hard time transforming it to assembly. The solution I found was to divide the tasks into multiple instructions. I first store the value at cur_min_idx address and tmp address in registers and then swap the values.

Possible Improvements to the Program

Several improvements could be made to this program. Just like other parts of the lab, the main improvement would be the use of less registers since they are only being used to store temporary intermediate calculation variables.