William Zhang
ID: 260975150

Friday, October 7, 2022

ECSE 324 – Lab 1 Report

In lab 1, the basics of ARM assembly are introduced, and subroutines and function calls are used. A subroutine consists of the caller and callee. The caller (the code calling the function) must move arguments into r0 through r3 and it should call the subroutine function using the BL instruction. On the other hand, the callee (the code that is called) must move the return value into r0, ensure that the state of the processor is restored to what it was before the subroutine call by popping arguments off the stack, and it must use BX LR to return to the code.

Task 1: Integer Division

Problem

The first task of lab 1 consists of implementing an integer division subroutine function using assembly. Only substractions and comparisons are allowed to find the remainder and the quotient given two positive integers. The subroutine has 3 arguments, and the implementation has to use r0 to pass the dividend, r1 to pass the divisor, and r2 to pass the address of the word in memory in which the result will be stored. Finally, the quotient and the remainder must be stored in a single word in the memory. The dividend and divisor values must fit in 16 bits because the result shifts the quotient by 16 bits to the left and each register can hold up to 32 bits. Consequently, the remainder and quotient values must be less than 16 bits and there are combinations which result in a 16-bit value of the quotient or the remainder. A value that takes up more than 16 bits will not fit in the register or the space allocated for the result.

Solution

The idea behind the solution is to subtract the divisor from the dividend until the remainder is smaller than the divisor. Every subtraction by the divisor increments the quotient register by 1. In the end, both the quotient and remainder are obtained in two registers. The solution is implemented using one subroutine function supported by three branch instructions. The subroutine function (dev) pushes the registers required for the operation into the stack, then initializes values for them. Then, the first branch (dev2) compares the values of the numbers and determines which branch to call given the result of the comparison. If the remainder is greater than or equal to the divisor, then the function will branch into ‘devloop’. If it is smaller than the divisor, then it branches to the ‘continue’ branch.

The ‘devloop’ branch subtracts the divisor from the remainder and increments the quotient register by 1. It then branches back into ‘dev2’. The continue branch, which finalizes the answer, merges the remainder and the quotient's value into the remainder register by shifting the quotient value by 16 bits. The program then stores the value into r4 (space allocated for the result) and makes r0 point to the address of the result. Finally, it pops the registers needed off the stack to return them to their initial values before the call and returns to the address where the function was called in the memory map.

Improvements

Possible improvements to the program include a reduction in the number of branches used for the subroutine as well as a more thorough adhesion to the constraints. For example, the three branches could be reduced to two branches by transferring the contents of ‘devloop’ into dev2 as dev2 is calling ‘devloop’ automatically if the program does not branch into the continue section. Thus, the existence of the ‘devloop’ branch is not required. Additionally, the constraints are technically not respected as addition is used in the program. A potential solution to the problem would be to subtract the negative of the number that should be added instead of adding the number itself.

Task 2: Iterative Insertion Sort

Problem

The second task of the lab consists of implementing the insertion sort algorithm iteratively using assembly. The insertion sort algorithm works by going through each value in an array in order, then comparing it with the values whose index is lower than it. All values in the sorted array that are bigger than the value being sorted are then shifted to the right by one

position and the value is inserted between the largest value which is smaller than it and the smallest value which is larger than it in the array. The process is repeated until all the values have been sorted. The implementation must match the code provided in the instructions, and the subroutine has two arguments. The register r0 must be used to pass the pointer to an array and r1 must be used to pass the length of the array.

Solution

The solution is implemented using one subroutine function and three branches. When insertionsort is called using bl, the program pushes the registers r2 through r6 into the stack. It then loads the value of r1 into r1 (length of the array) and multiplies it by 4 (the array is of type word, so the addresses of each node of the array are separated by 4 bits) using a logical shift to the left by 2 bits. Then, the constant value of 4 is moved into r2. The program then goes into the 'sort' branch, which adds the initial value of the address of the initial array to the address of the current index (in bits). If the value of r3 is smaller or equal to the length of the array in bits, then 'innerloop' is called.

The 'innerloop' branch loads the value at the current index and the value at the previous index into two registers, then compares them. If the value at the current index is greater or equal to the previous value, the reset branch is called. Otherwise, the values' positions are switched, and the index is decremented by 4 bits. The 'innerloop' branch will call itself until the current value is greater than all the values before it in the sorted array, or if it reaches the beginning of the array (the current value is the smallest value in the array), at which point it will call the 'reset' branch.

The 'reset' branch increments r2 by 4 (current index in bits), then compares its value to the length of the array in bits. If the value of r2 is lesser than the value of the length of the array, the 'sort' branch is called again. Otherwise, the registers are popped off the stack and the function returns.

Improvements

Multiple improvements could be made to the program. For example, the number of branches and the number of registers could be reduced and the program could be modified so that it is more "insertion"-like. The number of branches could be reduced as the reset branch's content could be moved into the 'sort' branch. The 'sort' branch could indeed be optimized since it doesn't have any procedures to go through if the comparison's result does not result in 'innerloop' being called. Also, the number of registers used could be decreased as we know that the initial memory address of the array is at 00000000 so we do not need to add the address of the array to the current value of j in bits to find the address of the current value. The two comparisons in 'sort' and 'reset' have the same function and could be merged. Indeed, both registers r2 and r3 have the same value as the value of r0 (the address of the array) is 00000000.

Secondly, the program could be modified so that it resembles the algorithm more closely. The implementation of the insertion sort algorithm currently looks a bit like a bubble sort as a value to be sorted drifts down one index at a time. To make the implementation more "insertion"-like, the 'innerloop' branch could be modified so that if the value at the current index is smaller than the value at the previous index, the current value is stored in another register. The value at the previous index is then moved up by one index. Then, the value that was stored is compared to the value at (index-2). Finally, when the value finds a value bigger than itself, it is inserted back into the array at the empty node since all the values greater than it have been moved up one position in the array.

Task 3: Recursive Insertion Sort

Problem

The third task of the lab is to modify the provided C code for the iterative insertion sort to make it a recursive function and to implement that function using assembly. The same input arguments as task 2 have to be used, but additional arguments can be added to the function. The program has to be implemented by following the subroutine calling convention and performing a function call by passing the LR and the registers on the stack.

The recursive algorithm consists of a base case, a recursive call, and a while loop in the C code provided in figure 1. The base case stops each recursion instance when the input argument for the length of the array is less to or equal than 1 since the code words using comparison. Then, the function is called recursively but for a decreasing length (by one) each time. Subsequently, the (n-1)th element of the array, depending on the length given as an input argument in the function call, is given to a variable. The while loop then goes through the elements in the array that are of a lesser index than (n-1) and moves them by one position to the right if they are greater than the element to be sorted. To go through the elements in the array, an index is used and its' value is decremented each time the while loop encounters a value that is greater than the

value to be sorted whose index is less than n-1 where n is the input argument of the function call. Finally, the value to be sorted is inserted at the final position (j+1) indicated by the index used to go through the elements in the array.

Solution

The solution to task 3 is implemented using one function and two branches. An additional argument is also needed and is used as an input argument to the recursive function calls instead of the initial length of the array. The function, 'insertionsort', pushes r3 through r8 and lr into the stack. It is important to push lr into the stack at the start because between the first BL call and each successive recursive call, the lr register content changes. By pushing and popping lr, the content of lr will be restored to what it was before the recursive function call.

The 'insertionsort' branch compares r3 (index register) to the constant number 1 and branches into 'exit' if r3 is lesser than 1. Otherwise, r3 is decremented by 1 and the function is called recursively. Finally, the index register r3 is multiplied by 4 and moved into r8. The register r8 contains the actual address of the value at the index passed as an input argument since it is known that the initial address of the array is 00000000. It then branches into 'innerloop'.

The 'innerloop' branch compares the value to be sorted with the value at the previous index. If the value to be sorted is greater or equal, the program branches into 'exit'. Otherwise, the values' positions are switched. and r8 is decremented by 4. Finally, r8 is compared to the initial address of the array (r0) to see if the while loop should stop. If the loop is required to stop (r8 <= r0), the 'exit' branch is called. The 'exit' branch pops r3 through r8 as well as lr and returns to the address indicated by lr.

Improvements

Possible improvements to the program include a reduction in the number of registers used by the subroutine. For example, it would be possible to delete r8 and use r3 instead as each function call will push and pop the registers off the stack, thus restoring r3 to the value it had before the function call. It is thus not required to copy the value of r3 into r8 so that the value of r3 can be conserved. Also, it is unnecessary to restore the value of r8 to 0 in the 'exit branch' as shown in figure 2 since the function will pop the registers and restore them to their previous values.

Task 4: Tail-Recursion Elimination

Problem

The fourth task of the lab is to modify the program from task 3 so that it only uses branching instead of function calls without saving and restoring states from the stack.

Solution

The solution to task 4 is to make a BL call outside the subroutine and instead of making function calls inside the subroutine, the program branches to itself. The 'insertionsort' branch uses r4 as an index register and iterates through the array while incrementing the index register for each pass. The 'innerloop' branch stays unchanged and has the same process as in task 2 and 3. The index register starts at 1 (the second element in the array) since the innerloop works using comparison between two elements in the array. When the index register is equal to the length of the array, the program exits the branch.

Improvements

The solution provided for task 4 is incorrect and needs to be modified. First of all, there must still be a BL call outside the subroutine and all the registers used should be set to 0 after the function call to make sure that the states are the same when compared to before the function is called.

Task 5: Noise Reduction with Median Filter

Problem

The last task of the lab is to implement a program that works as a 5x5 median filter for a 10x10 RGBA image. A median filter uses a nxm window to slide over the image pixels and selects the median value inside the window. An array of words containing 4 bytes is given and each byte of the word corresponds to a channel (RGBA). The median filter must be applied

to all four channels. The subroutine for the median filter takes one argument in r0 to pass a pointer to the first pixel of the image. It must also return a pointer to the first pixel of the result, but the filtered image must be stored in a different location in memory. The final image must be 6x6, the insertion sort algorithm must be used to find the median, and all four channels of the filtered image must be stored in a single word in the memory.

Solution

The solution to task 5 is incomplete. The code submitted allows the user to sort through the first 5x5 window but does not allow the program to differentiate between the channels and find the median of each channel. The program first calls the 'copyarray' subroutine, which pushes r4 through r12 onto the stack. It then branches into 'copy'.

The 'copy' branch copies each value of the input_image into a 100-bit space called "arraycopy" and increments the counter by 1. Every time a fifth value is copied, the address register is incremented by 20 bits (5 words). Once the counter reaches 25, the program branches into 'end'. The 'end' branch pops r4 through r12 and returns to the address indicated by lr.

The program then calls the 'insertionsort1' function. The subroutine first pushes the registers r4 through r12 and lr onto the stack since it uses the recursive insertion sort algorithm. It then sorts all the words. However, the values of the words are represented using signed representation so the sorting algorithm does not work for the purposes of the task unless the sign bit is removed.

Improvements

As the solution to task 5 is incomplete, many improvements can be made to the solution. The solution is only able to copy a 5x5 window onto an array and then sort the words in that array. Two 100-bit spaces and one 144-bit space are enough to implement the subroutine. First, to make the window move, the address of the input_image should be incremented by 1 everytime a median value is obtained for the resulting array. Secondly, to find the median value of each array, the starting index should be incremented by 13 (middle value) and then multiplied by 4 (LSL #4) to obtain the address of the median value.

Thirdly, to separate the channels, an 'and' gate should be implemented for each channel so that each channel value can be isolated and then sorted. For the first two bits of each word, a logical shift to the right by 1 can be made so that the sign bit becomes null. That would allow the sorting function to sort the numbers correctly. To fuse the median values of each channel back together, the first channel would be shifted to the left by one bit and each subsequent channel median value could be added as it is to the result as they are aligned. Finally, the value of the register which stores the address for the space allocated to the result array is incremented by 4.

The program's sequence starts by copying the 5x5 window into the 100-bit space. The register which tracks the address of the input_image is then incremented by 4. The 5x5 window is then copied onto another 100-bit space and the first channel is isolated and its values are shifted by one to the right to negate the sign. The window is then sorted and the median value is shifted back and stored in the 'result' array. The process is repeated for the three other channels. The process is repeated for each window until the value of the register which tracks the address of the input_image is 20 (post-increment). The window needs to move down by one and reset horizontally. The register which tracks the address of the input_image is then incremented by 20. When the register reaches the value of 224 (post-increment), the subroutine exits as the last window has been processed.

In conclusion, this lab introduced the basics of ARM assembly as well as the proper implementation of subroutine functions. Task 1 consisted of the implementation of a quotient and remainder calculator. The second task was the implementation of an iterative insertion sort algorithm. The third task was the implementation of a recursive insertion sort algorithm. The fourth task was the implementation of an insertion sort algorithm using branching only. Finally, the fifth task was the implementation of a 5x5 median filter for a 10x10 RGBA image.

Figures

```c
// Function to perform insertion sort on `arr[]`
void insertionSort(int arr[], int n)
{
    // base case (1 element (first) in array
    if (n<=1)
        return;
    // recursive call
    insertionSort(arr, n-1);
    // Starts with n = 2 and ends with n = length or array
    int valuetosort = arr[n-1]; //take value of nth element in array
    int j = n-2 // index of element right before nth element
    // while loop to compare and move elements that are greater than valuetosort
one position ahead
    while (j>= 0 && arr[j] > last) {
        arr[j+1] = arr[j];
        j--;

        arr[j+1]= valuetosort; // insert value in its correct position
    }
}
```

Figure 1: Recursive Insertion Sort in C

```
40 exit:
41 mov r8, #0
42 pop {r3-r8, lr}
43 bx lr
44 .end
```

Figure 2: Task 3 ‘exit’ Branch Code