ECSE 324: Lab 2

Theodore Janson, 260868223
Edin Huskic, 260795559

February 21, 2020

I. STACK

A. Problem and Approach

The problem was to implement the push and pop pseudo-instructions using other ARM instructions. To do this, used an array of numbers and pushed each one onto the stack. We started with a pointer to the first element in the list and a down counter initialized to the length of the list. The pointer to the first element in the list was loaded into R0. We then needed to push that element onto the stack: the stack pointer SP was decremented by 4 to update the top of the stack location and the value in R0 was stored in this location. Then, the pointer was updated and the process was repeated until the counter counted down to 0. At this point, the last element in the array was at the top of the stack. To pop the first element of the stack, R0 was loaded with the value contained at the memory address SP, which is the top of the stack. This address was updated by adding 4 to SP and the process was repeated with R1 and R2. So, in summary, an array of numbers A,B,C… were pushed onto the stack and then popped off in the order … C , B , A , which is LIFO.

B. Challenges Faced

The main challenge faced involved knowing whether or not to increment or decrement the stack pointer. Since the stack normally grows in the direction of decreasing memory addresses, we decided to decrement it when pushing and increment it when popping.

C. Possible Improvements

A possible improvement would have involved using the instruction LDMIA SP!, {R0-R2} to pop the values on the stack into those registers. This instruction works by loading R0 through R2 using the value at location SP and updating SP using IA (load and increment by 4 after). Similarly, STMDB SP!, {R0} could have been used during each loop pass to store the current value held by the list pointer on the top of the stack and then updating the stack pointer with DB (decrement before).

II. SUBROUTINE CALLING CONVENTION

A. Problem and Approach

The problem was to find the greatest element in a list of numbers by calling a subroutine using BL and passing arguments in R0-R4. The execution would then have to return from the subroutine using BX XL.

First the max register R0 is loaded with the first element in the list, and the subroutine loop is called using BL. A down counter is used to iterated through each element and is decremented during each pass through the loop. In each loop pass, the number stored in the max register is compared to the element loaded into the “current” register R1. The max register is updated if the current value is greater than it. R1 is updated by updating the pointer contained in R3 and copying it into R1. Once the counter had counted down to 0, the largest integer was stored in memory. The execution knows where to go back to when doing its final loop by using the branch and exchange instruction which contains the address for the last instruction, which is to store the max value into memory. Because few register required for the algorithm, none were passed to the subroutine on the stack. We also saved the state of the processor before the subroutine call by pushing the values contained in registers R2 and R0 by pushing their values onto the stack and then popping them back into these registers when the subroutine had executed. Parameters for number of elements and the first element were in R2 and R3. R1 was a local variable holding the current element being compared

to the current max. R0 was using to the return value (max).

B. Challenges Faced

The main issue with this section was implementing the same code in the previous laboratory using the new set of commands for subroutines that the team was unfamiliar with. Some coding experimentation was done in order to understand how the subroutine functions work and how the different registers react to this change. Once enough knowledge was collected, modifying the code to fit the requirements was effortless.

C. Possible Improvements

Since the code is based on an Assembly program given in the previous laboratory, there is not much room for improvements. A possible improvement would be to use less registers in order to make the code more efficient with its resource usage.

III. FIBONACCI

A. Problem and Approach

This part of the assignment consisted in developing an Assembly program which would compute the nth Fibonacci number for any given n. The program was tested mostly with small n's, as larger ones required large run times due to the high number of recursive subroutines.

After much trial and error, we decided to visualize this problem using the idea of a call tree, as seen in Figure one.

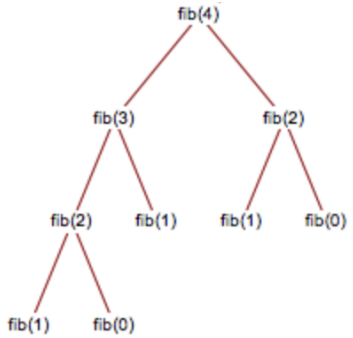

Figure 1: Fibonacci call tree

Our solution for computing the nth Fibonacci number consisted in doing a sort of in order traversal, in which the node fib(n-1) would be visited, then the root node and then the node fib(n-2) would be visited. Whenever the execution would reach a leaf node, we would call in the Fibonacci terminating condition (fib(1) = fib(0) = 1) by counting the number of these occurrences in a register.

The program starts by initializing the contents of two registers R0 and R3 with the desired fibonacci number and with 0 as a counter. The fibonacci subroutine is then called. Consider the 4th fibonacci number case. The program then branches to the fibonacci subroutine and places the address of the subsequent (final instruction) into the LR so that the program can return from the subroutine. The subroutine starts by pushing R1, R2, and the link register onto the stack. R1 and R2's contents are currently not initialized nor important, but LR allows the execution to return from the last subroutine call. The value of the root node in R0 is copied into R1 and R2, which are used to hold the values of the left and right child nodes respectively. The execution evaluates the left node first by testing if the value of a current node (R0) is less than or equal to 2. In such cases, the next recursive call of the subroutine would reach a terminating case, so the counter is to be incremented. With R0 = 4, this doesn't happen, so the execution continues down the left of the tree by decrementing R1 by 1 and then copying the contents of R1 into R0 as the child node is now viewed as a parent node. The subroutine is called once again, storing the new value of R1 and LR onto the stack. This means that the current node's value is stored in R0 to be used as a parent node and in R1 to be used as a child node relative to the nodes above it. Saving LR means that we can go back up the tree to evaluate the right child nodes. The

execution continues down to left side of the tree until it reaches a base case, in which case the counter is incremented and the top 3 contents of the stack, R1, R2, and LR are popped and the values contained on the stack are placed back into the registers and into the PC. This allows for the execution to go from fib(1) back to fib(2), so that fib(0) might be evaluated, in which case the same process occurs using R2 instead of R1. Once the nested subroutines are called, the execution works its way up the tree but popping the corresponding stored LR values into the PC. Eventually, the execution returns from the right side of the tree and terminates on the parent node as both its children have been evaluated, in which case the first LR push is popped and we return to the main. The final sum is then stored.

B. Challenges Faced

The program had to be worked on very gradually by trial and error to see get a good understanding of when and which registers should be pushed onto to the stack, when they should be popped, etc. This required running incomplete code and observing the content of the registers as we tried to follow the call tree. To do this, we used the visUAL program which allowed for the program to be run and de debugged on our own computers, rather than on the Altera board. We were having trouble with testing the program using the board due to the long wait times, so the visUAL program resolved this with its fast runtimes. We also had wrote pseudocode that would implement fib(3) without subroutine calls (ie: lines of code, rather than a loop) to determine when to push the LR onto the stack, and when to pop it into the PC. Another challenge consisted in finding a way to save the values of the child nodes and then make a child node a root node when moving down the tree. This was resolved by pushing the child node values onto the stack and then copying the value of the current register (R0) into the values for the children nodes (R1, R2) so that these could be decremented by 1 and 2 as the Fibonacci algorithm requires.

C. Possible Improvements

A possible improvement could involve rewriting the code so that rather than using a register to keep track of each time a leaf node was encountered, the value of 1 would be returned by pushing it onto the stack when visiting a leaf and then returned it to the node that called the leaf when moving up the tree by popping it off the stack. A parent node would then contain the sum of number of the number of leaves in branches to.

IV. PURE C

A. Problem and Approach

In this section, we had to complete some C code to compute the max value of an array. To do this, we iterated over every element in the array using a for loop which stopped when it counted the number of elements in the array. We needed a way to find the number of elements even if it wasn't specified in the array declaration. The number of elements was computed using the fraction sizeof(array) / sizeof (array[i]). The operator sizeof(a) determined the bit size of the parameter. Since the array size S can be calculated with S = N * sizeof(a[i]), the number of elements N can be obtained by solving for it. So while looping from 0 to N, the max value for the array is updated to the current value a[i] if the latter is greater than it. Finally, this max value is returned.

B. Challenges Faced

The main challenge was determining how to find the number of elements in the list, which was described above. Also, we had to solve an error which required declaring the iterator integer i outside the for-loop, rather than in it, as one does in Java.

C. Possible Improvements

A possible improvement would be to adapt the code to return the maximum value of any input array.

V. CALLING AN ASSEMBLY SUBROUTINE FROM C

A. Problem and Approach

The goal of this section was to implement a given Assembly subroutine for computing the maximum of two values into the previous C code used to compute the maximum value of an array. For this to work, both the C program and the Assembly program were imported into the project for the code to work. Once imported, the "*for"* loop was modified in the C file was modified in order to utilize the given Assembly subroutine function rather than the iteration code in C. The Assembly function was also imported at the start of the C code for the function to be used.

B. Challenges Faced

The main challenge was determining how to find the number of elements in the list, which was described above. Also, we had to solve an error which required declaring the iterator integer i outside the for-loop, rather than in it, as one does in Java.

C. Possible Improvements

Just like the previous part, a possible improvement for this code would be to obtain the maximum value of any input array.