ECSE 324

Lab Report-1

Faiyad Irfan Hares

Part 1.1a

Description: This part of the lab asked us to develop an assembly program that can find the nth Fibonacci number by using an iterative algorithm.

Approach used: The algorithm in this part was achieved using an iterative method. I used a loop to successfully program this part since the c code provided to us also has a for loop. The nth Fibonacci number is calculated using the formula f[n] = f[n - 1] + f[n - 2]. The for loop in the C program provided to us is as follows:

```
for(i = 2; i <= n; i++)
  {
    f[i] = f[i - 1] + f[i - 2];
  }
  return f[n];
```

The same instructions given in the For loop of the C program can be written as a simple loop in assembly language. The loop that I wrote is:

```
LOOP:

 CMP r3,r4  // compares the iteration value to n(condition for loop)
 STR r0,sum
 BGT END    // if i-n>0, loop terminates
 MOV r5,r0  //stores the original sum r0 in a temporary register
 ADD r0,r1,r2 // adds f(n-1) and f(n-2) and stores it in r0
 MOV r2,r0   //stores the new sum in n-1
 MOV r1,r5   // stores the original sum in n-2
  ADD r3,r3,#1
 B LOOP
```

The first part of my loop was specifically written to terminate itself if the iteration going on is greater than n. If the loop does not terminate, then the program adds f(n-1) and f(n-0) and stores it in r0. This new sum in r0 is then stored in r2. The original sum is stored in r1. After that the loop is repeated till i becomes equal to n and the number is found.

Challenges faced: This was the first time I used a loop in assembly language, so it took some time and lots of trial and error to finally figure out how a loop works.

Possible Improvements: In this program I used registers to temporarily store the new sum and original sum. The program might be improved by removing the use of these temporary registers.

Part 1.1b

Description: This part of the lab asked us to develop an assembly program that can find the nth Fibonacci number using a recursive algorithm.

Approach used: To implement the program for this part, I used a recursive subroutine as instructed. I had to carefully write my program so that it follows the proper subroutine calling convention. Recursion involves calling a method or a function within itself. The c code for this recursive approach was provided:

```
int Fib(int n) {

 if (n <= 1)
     return n;
  else
     return Fib(n-1) + Fib(n-2);
}
```

As can be seen above, the function called Fib, is called within itself to calculate the nth Fibonacci number. This can be implemented in assembly language using the subroutine calling convention and by storing some of the values on the stack. The program I implemented was as follows:

```
 FIB:                    STMDB SP!, {R1, R2}    // save the values of R1 and R2 on the stack
        CMP R3, #2          // compare R0 to 2 (i.e. base case)
        BGE RECUR           // jump to our recursion instructions if R0 is not less than 2

BASE:                    MOV R0, R3          // move 1 into R0
        B DONE             // jump to DONE

RECUR:                   STMDB SP!, {R3, LR}    // save the values of R3 and LR on the stack
        SUB R3, R3, #1      // subtract 1 from R3
        BL FIB             // call the FIB subroutine Fib(n - 1)
        MOV R1, R0          // move the return value R0 to R1
        LDMIA SP!, {R3, LR}    // pop the values of R3 and LR off the stack
        STMDB SP!, {R3, LR}    // save the values of R3 and LR on the stack
        SUB R3, R3, #2      // subtract 2 from R3
        BL FIB             // call the FIB subroutine Fib(n - 2)
        MOV R2, R0          // move the return value R0 to R5
        LDMIA SP!, {R3, LR}    // pop the values of R0 and LR off the stack
        ADD R0, R1, R2      // add R1 and R2 and store the result in R0

DONE:                    LDMIA SP!, {R1, R2}    // pop the values of R1 and R2 off the stack
        BX LR              // return
```

As you can see that I begin the program by defining my subroutine FIB. I later call this subroutine in the block of code that is defined by the label RECUR. In my subroutine, I compare the value of n to the base case 2. If n is more than 2, then the program jumps to the recursion. In the recursion I used the stack to temporarily store my values so that when I call the subroutine FIB, it pops off the values from the stack. The formula for this program is the same as the formula used in the iterative approach. However, the values of f(n-1) and f(n-2) are stored on the stack. After these values are stored on the stack, the FIB subroutine is called on these values. The Fibonacci number is then calculated and stored in register R0.

Challenges faced: This program was extremely difficult to implement since I used a recursive subroutine. It took a lot of trial and error to find the correct program where all the rules of the subroutine calling convention were followed.

Possible Improvements: Since I used the stack to store values in this program, it is already quite efficient and much faster than heap-based memory allocation. Th program I wrote can definitely be more organised and made a bit simpler, but when it comes to execution, I do not think there is a more efficient way to do this.

Part 2

I did not attempt this part of the lab.

Part 3

Description: In this part of the lab, we were asked to implement a bubble-sort algorithm in assembly language that sorts a given array in ascending order.

Approach used: The c code for the bubble sort algorithm that was provided to us involved the use of nested for loops to compare and store the numbers in the array in ascending order:

```
for (int step = 0; step < size - 1; step++) {
 for (int i = 0; i < size - step - 1; i++) {


  // Sorting in ascending order.
  // To sort in descending order, change ">" to "<".
  if (*(ptr + i) > *(ptr + i + 1)) {
   // Swap if the larger element is in a later position.
   int tmp = *(ptr + i);
   *(ptr + i) = *(ptr + i + 1);
   *(ptr + i + 1) = tmp;
  }
 }
}
```

The program I used to implement the bubble sort algorithm has 2 loops to compare the values in the array:

```
LOOP_ONE:        MOV R7, R6                    // R7 holds loop counter for LOOP_TWO
                 SUBS R6, R6, #1               // decrement the loop counter LOOP_ONE
                 BEQ END                       // end loop if counter has reached 0
                 ADD R3, R4, #8                // R3 points to the first number
```

```
                LDR R0, [R3]            // R0 holds the first number in the list
                ADD R5, R4, #12         // R5 points to second number
                LDR R1, [R5]            // R1 holds second number in the list
                B LOOP_TWO                      // branch to LOOP_TWO

LOOP_TWO:   SUBS R7, R7, #1         // decrement current loop length
                BEQ LOOP_ONE            // end loop if counter has reached 0
                CMP R1, R0                      // check if R1 is less than the R0
                BLT SWAP                // branch to SWAP if R1 is less than R0
                B SHIFT                         // branch to SHIFT
```

The 2 loops are nested and work together to copy 2 values from the array, compare them and then store them in ascending order. The first loop is used to hold the first 2 numbers in the list in 2 registers. The second loop compares the value in these 2 registers. If the first value is greater than the second value, then the program branches to SWAP, which basically ensures that the numbers are moved around in the ascending order. If the first value is less than the second value, then the program branches to SHIFT. SHIFT ensures that the third and second value of the list is stored in 2 registers, and then branches to the second loop to compare these 2 values. This is repeated until the whole list is sorted in ascending order.

Challenges faced: The 2 loops were quite difficult to implement since I have never implemented nested loops in assembly language before. The program was also quite confusing since it involved several steps and comparisons before the whole list was sorted.

Possible improvements: The program I used compared each value with the next till the ascending order was achieved. This could amount up to a very large number of comparisons if large arrays are involved. A more efficient program would keep the required comparisons to a minimum.