LAB #1
ECE 3140/CS 3420

Divyansh Garg (dg595) and Caitlin Stanton (cs968)

INTRODUCTION

This lab assignment was split into three parts that overlapped in functionality and implementation. Part 1 focused on programming the FRDM board to allow its LED to blink the Morse code equivalent for a value stored in a certain register. The point of programming this was to learn more about Assembly and its interactions with the registers used to hold data. Part 2 built on this by implementing this Morse code translation as a function, giving us an opportunity to use the calling conventions established by ARM. Part 3 added another layer of knowledge about ARM and Assembly by making us use recursion and loops to perform a Fibonacci calculation and display the result using the Morse code function from Part 2.

PART 1

More specifically, an integer value between zero and nine was stored in register R0. This value had to be read and then interpreted in Morse code through a series of dots and dashes. Dots were a singular “blink” by the LED, and dashes were an LED signal that lasted multiple times longer than that of the dot. A delay between the dots and dashes needed to be incorporated as well which was same as the duration of a dot. Also, as the frequency of the LED too high for different signals to be perceptible by the human eye, we needed a loop for some time interval where no operations were performed and the LED remained in a fixed state.

The implementation for Part 1 was written through Part 2, which went into more detail for specific use of calling conventions, rather than just lines of Assembly meant to manually interpret the value in R0.

PART 2

The purpose of this section of the lab was to delve deeper into Assembly and ARM calling conventions so that we would be able to implement the Morse code interpretation of Part 1 as part of a function. We made sure that the caller first used caller-saved registers R0 through R3 before accessing other registers, as those would have to be added to the stack (for our later function ***fib***, this was the case with R4 and R5). Also, parameters passed to the callee were done using registers R0-R3 in order, although we only R0 was used for ***MorseDigit()*** function.
In addition, our callee (the function itself) always outputted its result to R0, so we made sure that R0's value was never corrupted through any unrelated operations that would overwrite the function result. To stay moving consistently through the program, the LR register was used by pushing and popping it from the stack so that the instruction to return to after every branch, loop, or helper function was updated.

After looking through the table for Morse code (Figure 1), we noticed a pattern for the dots and dashes that we decided to take advantage of when implementing the ***MorseDigit()*** function.

Digit	Code
1	dot dash dash dash dash
2	dot dot dash dash dash
3	dot dot dot dash dash
4	dot dot dot dot dash
5	dot dot dot dot dot
6	dash dot dot dot dot
7	dash dash dot dot dot
8	dash dash dash dot dot
9	dash dash dash dash dot
0	dash dash dash dash dash

Figure 1 -- The Morse Code signals for numbers 0-9

For numbers from one to five, dots would begin the signal. The number of dots corresponded to the number itself; for example, the number three would start off with three dots. These dots would be followed by dashes, the number of which equalled the difference between five and the number, making the signal a length of five dots and/or

dashes. Sticking with the number three, this meant that there would be two (5 - 3) dashes, giving three the complete signal: dot dot dot dash dash.

For numbers from six to nine (including zero), dashes would begin the signal. The number of dashes was equal to the difference between five and the number itself. If the number in this case was eight, then there would be three dashes because the difference between eight and five is three. After these dashes, the number of dots that would follow was found by subtracting the number of dashes from the total number of five signals. Therefore, this would make the full signal for eight dash dash dash dot dot, since the number of remaining signals was two.

Our function, called ***morseDigit***, used several different helper functions and loops to display a single digit in Morse code according to the above patterns.

In order to have the program remember where to return, the value of the LR register was first pushed onto the stack. Then the value of R0, which stored the single digit, was compared to five. If it were greater than five, it jumped to a branch labeled ***n_gt5***, which handled the signal as described in the paragraph for numbers six, seven, eight, nine, and zero. For this case, registers R0 and R1 stored the number of dashes and dots, respectively. The number of dashes was calculated in the loop ***loop_dash2***, which called the function ***dash*** (which would turn on the LED and delay it for a time long enough to be considered a dash) as long as the value of R0 was greater than zero and then subtracted one from R0 to go through the loop again.
Once the value of R0 was less than or equal to zero, it branched to the loop ***loop_dot2***. This loop similarly compared the value of R1 to zero and branched back to itself as long as R1 was greater than zero. If this was the case, it called the ***dot*** function to turn the LED on for a short blip, subtracted one from R1, and returned to the top of the loop. When the value of R1 reached a value that was less than or equal to zero, it branched to ***done***, which signalled the end of the Morse code by popping the value of LR from the stack and branching back to that instruction in the program.

If the branch didn't take ***morseDigit*** to ***n_gt5***, then it was handling single digits that were either one, two, three, four, or five. The values for the number of dots and dashes were stored in R0 and R1, respectively, based on the algorithm described earlier in this section. The program first was taken to ***loop_dot1*** and performed comparisons and instructions similar to that of the loops in ***n_gt5***. The value of R0 was compared to zero: if it was greater than zero, then ***dot*** would be called to signal an LED dot, one would be subtracted from R0, and ***loop_dot1*** would be branched to again; if it were less than or equal to zero, then it branched to the next loop ***loop_dash1***. Here, the value of R1 was

compared to zero: if it was greater than zero, then ***dash*** was called to signal an LED dash, one was subtracted from R1, and ***loop_dash1*** was branched to again; if it were less than or equal to zero, then it branched to ***done*** that was described earlier.

The ***dot*** and ***dash*** functions also used a function called ***space*** that we wrote to delay the LED appropriately, so that the difference between a dot and dash could be recognized easily. This function was also used to distinguish the dots and dashes.

We tested this code, used to solve for both Parts 1 and 2, in ***main*** by writing values to R0 for all single digits between zero and nine and then calling ***morseDigit***. These calls were spaced out with several calls to ***space***, so that each number had its own distinct signal for its Morse code value and could be easily recognized by the tester.

PART 3

This part of the lab used the function ***morseDigit*** from Part 2 to show the Morse code signal for the first seven Fibonacci numbers (0, 1, 1, 2, 3, 5, 8). Based on the nature of the Fibonacci sequence, where each number is based on the sum of the two Fibonacci numbers before it, it was necessary to use the stack to implement recursion. An int (n) was stored in R0, and then the nth Fibonacci number was calculated and displayed in Morse code.

To start, we pushed R4, R5, and LR onto the stack, to preserve local variables (R4 and R5) as well as instruction for the program to return to (LR). The value of R0 was compared to one, which then led to three potential cases that could be taken: R0 was equal to one, less than one, and greater than one.

The first two of the three cases were the base cases. For values of R0 that are less than one, the value of zero was stored in R0. For values of R0 that are equal to one, the value of one was stored in R0.

For values of R0 that were greater than one, the program branched to ***gt1***. R4 was given the value of R0, to save n, and then one was subtracted from R0 and ***fib*** was called so that the Fibonacci value of n-1 could be calculated. After fib(n-1) was calculated, that output (in R0) was stored in R5. Since R4 had saved the original value n, the difference between R4 and two was stored in R5. The program then branched back to ***fib*** to calculate fib(n-2). To compute the final value of fib(n), the values of R0 (which has fib(n-2)) and R5 (which has fib(n-1)) are summed and stored in R0. The ***fib*** loop essentially ran through this case for as many times as possible until it was able to

enter the base cases, at which point the results of each ***fib*** call would sum to the Fibonacci number.

For testing the code, we wrote calls to ***fib*** and ***morseDigit*** functions in ***main*** for different values of R0. We used values like -20, -1, 0, 1, 4, 6, etc. These calls were spaced out with several calls to ***space***, so that each output could be easily recognized by the tester.

We use the stack to save R4, R5 and LR at the beginning of ***fib*** and recover them at the end of it. R4 contains the value of n before a nested function call and is recovered after the call returns. This is needed to calculate both fib (n-1) and fib (n-2). R5 contains the the value of fib (n-1), which is needed after the call to fib (n-2) returns to calculate fib (n). LR is needed to be saved to return to the original caller after a nested function call. For describing the usage of the stack for ***fib***, we work with an example n = 3.

LR
R5 (0)
R4 (0)

Fig 2: fib(3)
Stack before the 1st call to fib(2)

When we first call fib(3), we save LR, R5, R4 to stack with R5 and R4 being uninitialized and we can assume them to be 0, as in Fig 2.

LR
R5 (0)
R4 (0)
LR
R5 (0)
R4 (3)

Fig 3: fib(3) -> fib(2)
Stack in nested call to fib(2) and before the 1st call to fib(1)

Then we jump to ***gt1*** and save 3 to R4, and perform a call to fib(2), where we again save LR, R5 and R4 as in Fig 3.

LR
R5 (0)
R4 (0)
LR
R5 (0)
R4 (3)
LR
R5 (0)
R4 (2)

Fig 4: fib(3) -> fib(2) -> fib(1)
Stack during the 1st call to fib(1)

We again jump to ***gt1*** and save 2 to R4, and perform a call to fib(1), where we again save LR, R5 and R4 as in Fig 4.

LR
R5 (0)
R4 (0)
LR
R5 (0)
R4 (3)
LR
R5 (1)
R4 (2)

Fig 5: fib(3) -> fib(2) -> fib(1)
Stack during the 2nd call to fib(1)

Now, we finish the 1st call to fib(1) and store the result in R5, and pop the last 3 values from the stack in fig 4. We again perform a call to fib(1) and save LR, R5 and R4, as in Fig 5

LR
R5 (0)
R4 (0)
LR
R5 (1)
R4 (1)

Fig 6: fib(3) -> fib(1)
Stack during the 3rd call to fib(1)

We finish the 2nd call to fib(1) and also end the call to fib(2), while popping the last 6 elements of the stack in Fig 5. We then save fib(2) to R5 and perform the last call to fib(1). LR, R5 and R4, are stored during the call as in Fig 6.

After we finish the call, popping the last 3 elements of Fig 6, we add R5 to the output (R0) to get fib(3). Finally, we clear the stack and return to the caller

EXTRA CREDITS

We tested our ***fib*** function for inputs like 20, and were satisfied to find our implementation worked for n > 6. For displaying multi-digit numbers in morse code, we created a function called ***morseMulti***. Respecting the calling conventions, we used R0 to input n and saved LR in the stack.

We had two major problems. First, extracting the digits from the number and second displaying it. For the 2nd part, as we extracted the digits starting from the lower places and then going to higher ones, we needed to reverse the order for displaying them. For e.g. for n = 32, we extracted 2 then 3, but we wanted to display 3 then 2 in order.

We used the stack to accomplish this. We initialized R1 to -1, and added it to the stack as a flag to signal the end of a number. For the 1st part, we added the extracted digits to the stack and for the 2nd part we popped them from the stack to display them until we hit -1.

For extracting the digits from the number, we worked out an algorithm to accomplish it.

We use ***SDIV*** instruction to divide R0 by R3 and store it in R5 to find (n/10). For calculating n mod 10 = n - 10 * (n/10). We multiply R5 by R3 and store it in R4 and then subtract R4 from R0. We store this result in R4. This represents the digit at the lowest place in the number. We then move it to R1 and store it in the stack. A loop is used to repeat this procedure on n/10, until the remaining number is 0.
For displaying the number, we pop from the stack and display the digit with a call to ***morseDigit***. We then call ***space*** to separate out the individual digits. And repeat this operation in a loop until we hit -1, signifying the end of a number.

We test this out in ***main***, with calls to morseMulti with values like 23, 71, 100, 245, etc.

WORK DISTRIBUTION

Before programming, we sat down and interpreted the lab assignment and identified the various concepts and coding constructs that would be needed for each part (i.e. recursion and loops for Part 3). After determining a foundational design for each part, we relied on pair programming to allow us both to contribute equally to the project and learn the appropriate Assembly and ARM concepts along the way. One of us would be typing the code while the other was researching appropriate instructions and looking through notes. As we programmed each part, we went back and wrote test calls for every potential number in the range of valid operands (for Part 1/2, numbers zero through nine; for Part 3, numbers zero through six). We walked through every line of the code for some of these test calls, to ensure that calling conventions were followed in addition to getting the right answer.

Since we used pair programming, we often were in the same space while coding and didn't need too many tools to share code. We did, just in case, establish a GitHub repository for version control and for us each to have equal access to the code for each part. Whenever we met up to code, the timing was established at the previous time we met so that it coordinated with both of our schedules.