ECE 3140 / CS 3420 Embedded Systems
Lab 1 Report
By: Matthew Daniel and Giancarlo Pacenza

Project Overview

Part 1 (includes design, coding, code review, testing, and writing.

For part 1, since the program uses pauses throughout, we started by creating a DELAY function which runs through a long loop from EEEE to 0 and then branches back. That delay was not long enough, so we created another function to then loop through this initial delay function a few times. We tested this function by turning the LED on, calling the delay loop, and turning the LED off. The coding of this part was where we had the most trouble. We were unsure of the exact syntax to create a loop and branch. We checked online sources and the lecture materials, but they didn’t specify that the loop cannot be indented for the branch name. Once we calibrated the program we called DOTDELAY, we then created the LEDDOT and LEDDASH programs. Both of these programs simply turn the LED on with LR LEDON, make either 1 or 3 (1 for dot, 3 for dash) calls to the delay function, and turn the LED off with LR LEDOFF. Once we wrote these, we were very careful to keep testing to make sure the delays and calls were working as expected. The coding of this was fairly simple and very similar to the first DELAY function.

At this point, we realized that all digits in morse code required sequential calls of either dots or dashes. To simplify the problem, we created two functions called DASHLOOP and DOTLOOP. We pass the number of calls for dash and dot calls in R0 as parameters to these functions. The functions loop through the number of calls and output the given number of dashes or dots respectively. Finally, we created a bunch of compares with 0 through 9 which branch to the according output. If the number is not an integer, it would branch back to the load register because it would not make sense to output anything but 0 through 9. If the code matches R0 to an integer, it branches to the according script which calls the correct number of dashes and dots in the correct order. We tested every single integer to make sure that the outputs were correct and working as expected.

Part 2

Part two was fairly simple in that we created a function from the script in part 1. That actually led us to encounter a few errors in our first part which we then corrected. We accidently popped the stack to R0 twice in the code, which led us to have the program work once, but not twice. We tested by checking a few different numbers similar to in part 1. This whole process was fairly straight forward (aside from our errors) and worked well after we fixed our mistakes.

Part 3

We used the template given in the lab 1 manual to start the code. The coding was fairly straight forward since it followed the same pattern as the C code. The function started by comparing the input with 0 and 1 and branched to the according output script. If the input is neither 0 nor 1, the program then subtracts one, pushes R0, calls fib again, saves the output of fib as R1, and then pops the stack to R0. Again it subtracts one and calls fib, and then finally adds the values of fib (x -1) and fib (x-2) and saves it to R0.

Using the stack in this problem is extremely important not only for when the program is called, but also inside the program when calling itself. When it calls fib within the program, it needs to first save R0, call the program, and then save R0 (which is a new value modified by fib) to another register, and finally pop the initial R0. This is because fib uses R0 twice to call fib (x-1) and fib (x-2) and needs to keep the value. We had a few issues with push and pop because we forgot to push and pop R0 when calling fib within the function. After debugging a few times we were able to get it working as expected. We tested the code for R0 = 0 to 6 on the board where we called fib and output onto the LED on our board. We also used debugging mode to check the values higher than 6 to be positive our program was working as expected.

Extra Credit

This problem was a lot less straightforward than the others, but we wrote out a plan on how to solve it. The initial plan was to have a function that subtracted 10^(n-1) to find the number of digits, and then call another function which outputs each digit, but we found that to be difficult. There is still code from that in the file to show how we first approached the problem, but in the end we found a much simpler solution. We decided to instead create a function called NUMDIVIDE to loop through the input number by dividing by 10^n and finding the remainder and pushing that to the stack. The program loops through until the division outputs 0 which means we have completed all of the numbers. Then the program runs a loop through each of the stack pushes and outputs the value to the MorseDigit in the correct order. We tested this program with the board as well with a few different 3 and 4 digit numbers to have confidence that this works!

Final Thoughts on code

We were happy with how our code turned out. One thing we should have done differently is making more comments throughout our code to make sure it is easy to follow. For simple code such as this, it is less important, but in the future we plan to write more helpful comments. Overall, we didn’t have too many errors in our code, but did have a major issue where our board essentially broke after we finished, and are in the process of getting it replaced by the TA’s.

Work Distribution / Collaboration

For this lab, we mainly worked together while coding. When we met the first time, we had both read the manual and were ready to start coding. Matthew Daniel has a windows machine so we are using his computer during these labs since the virtual machine is very slow and prone to issues. Matt did most of the actual typing, but we worked together while coding. We worked through the problems together helping each other along the way to get to our solution. We communicated through text and email to meet up and share the lab documents.