ECSE 324 - Lab 3 Report

Alexander Liu

March 24th, 2024

Part 1: Calculator

Code Overview

My calculator starts with a brief setup where 0's are written to all 6 HEX displays and the result variable is instantiated to 0. Then, it polls the button for a button press. Once a button is pressed, control flow will be diverted to the subroutine corresponding to the button. This subroutine disables all the other buttons and polls for a release. Once the release is detected, it conducts the rest of its operations which include computing the result and updating the HEX displays accordingly. As a side note, the result of my calculator is permanently stored within A4. Therefore, my subroutines do not use A4 as a regular argument register since I do not want to overwrite it unknowingly.

Performance Analysis

The main concern when it comes to performance for the calculator is the necessary conversion form hexadecimal to decimal or BCD (binary-coded decimal). I resorted to using the double-dabble algorithm (see sources below for more information) to tackle this issue. My implementation of the double-dabble algorithm, *hex_to_bcd*, adds an additional 40 lines to my calculator file's code size. In terms of actual performance, I have optimized it given the specifications in the lab document. Seeing as how the largest hexadecimal input we can encounter is 0x1869f, we can simply consider the maximum number of shifts we have to perform to be 17 instead of shifting the entirety of the register which would require 32 shifts. Therefore, the number of instructions executed will be 1 (branching to the subroutine) + 8 (pre-loop) + 28 (loop body) * 16 + 7 (on last iteration of loop when index = 16) + 1 (post-loop) = 465 instructions every time we need to convert the hexadecimal value to BCD. These instructions also require memory accesses, so we are also executing 3 (push 3 registers onto the stack) + 465 (instructions fetches) + 3 (popping 3 registers off the stack) = 471 memory accesses for each conversion. In other words, the work required after every arithmetic operation performed by the calculator to convert the result from hexadecimal to BCD is fetching and executing 465 instructions. Therefore, displaying the result of the calculator in base 10 is more costly performance wise than simply displaying the result in base 16. The takeaway here, though, is that the extra work remains constant and does not scale up as the input increases. Thus, this performance loss can be considered minimal especially since the processor operates at a very fast frequency.

To visualize this performance discrepancy more clearly, I will provide the number of instructions executed and memory accesses performed by the *hex_to_bcd* subroutine for various inputs. The table below serves showcase how displaying in base 16 would be more efficient (since we would not have to perform any additional work) and how the runtime of the conversion algorithm stays constant regardless of the size of the input. The breakpoints were inserted at the first and last instructions of the *hex_to_bcd* subroutine which handles the conversion from hexadecimal to BCD.

Number to display	Converting to Base 10 in *hex_to_bcd*		**Note:** The values here are 464 rather than the previously computed 465 (and 470 rather than 471) since we did not count the branch instruction to get to the subroutine.
	# Instructions Executed	# Memory Accesses	
99,999	464	470	
1,234	464	470	
-1	464	470	
-99,999	464	470	

** For the purposes of this report, fetching an instruction is counted as a memory access.*

Sources

- https://www.realdigital.org/doc/6dae6583570fd816d1d675b93578203d ← My implementation was inspired by the code snippet provided on this website.
- https://www.youtube.com/watch?v=eXIfZ1yKFlA ← I watched this video to gain basic understanding on how the double-dabble algorithm works.
- https://en.wikipedia.org/wiki/Double_dabble ← Yes, I used Wikipedia... to practice various examples.