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.

**Part 2: Timers and Interrupts**

**Code Overview**

I start by displaying the initial stream of characters which are the elements in the *CURR\_ROTATION* byte array on the HEX displays as well as the LEDs. Then, setup is required to enable interrupts on the private timer and the pushbuttons. Finally, we end up in the main body of the program, *IDLE*, where the pushbuttons interrupt flag, private timer interrupt flag, and switches are polled. Switches enable characters to change within the stream. For example, turning switch SW0 on will flip the character at position 0 inside *CURR\_ROTATION*.

**Performance Analysis**

The subroutine responsible for servicing interrupts from the timer and the pushbuttons, *SERVICE\_IRQ*, spans only 18 lines excluding comments. It calls 3 subroutines, *ARM\_TIM\_clear\_INT\_ASM, ARM\_TIM\_IS*, and *KEY\_ISR,* that are 6, 5, and 8 lines respectively. This brings the total code size to 37 lines. Effectively, running *SERVICE\_IRQ* once executes 33 instructions with slightly more memory accesses since we manipulate the stack. Given its small size, the total time spent within it is expected to be virtually insignificant. The switches are polled continuously in *IDLE*. Considering that the subroutine responsible for polling the switches, *check\_switches*, is 27 lines long and executes 3 (pre-loop) + 20 \* 10 (loop body) + 2 (last iteration of loop) + 1 (post-loop) = 206 instructions, the time spent within this subroutine is expected to be significant given the frequency at which we loop through *IDLE*. Finally, the user code (everything else in IDLE apart from *check\_switches*) takes up the other half of the *IDLE* block at 28 lines. It polls the interrupt flags and executes several subroutines if any of the flags is raised. Therefore, the code size of the user code is a lot larger than the 28 lines contained in the *IDLE* block. The number of instructions executed, and memory accesses are expected to be far greater than 28 as well, though the frequency at which the interrupt subroutines are ran depends on how often interrupt flags are raised. Therefore, we can establish that there is a proportional relationship between time spent servicing interrupts and frequency of the interrupts as well as time spent in the user code and the frequency of the interrupts.

To showcase this, I have provided data on instructions executed and memory accesses for *SERVICE\_IRQ*, *check\_switches*, and *IDLE* (excluding *check\_switches*). The table below contains the mentioned data and showcases the trend that the time spent servicing interrupts (i.e. in *SERVICE\_IRQ* and subroutines in *IDLE*) increase as the number of interrupts increases. This can effectively be observed through varying the frequency of the private timer (lower frequency = more interrupts = more time spent servicing interrupts). The lower the load value and prescaler bits are, the more frequent interrupts will be and vice versa. My approach to gathering data was to insert a breakpoint at the first instruction of *SERVICE\_IRQ* to find how many instructions are executed between the *\_start* segment and the first timer interrupt. Then, I subtracted the number of instructions in the *\_start* block to find how many of them can be attributed to polling the switches and user code respectively by using proportions. Lastly, I counted the number of instructions executed in the *SERVICE\_IRQ* and user code associated to handling interrupts and factored them into the data as well. Thus, the data provided can give a good idea on the amount of time spent in each block.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Effective Frequency | *SERVICE\_IRQ* (interrupts) | | *check\_switches* (polling switches) | | *IDLE* (user code) | |
| # Instructions Executed | # Memory Accesses | # Instructions Executed | # Memory Accesses | # Instructions Executed | # Memory Accesses |
| 20 MHz | 24 | 54 | ~ 129,000 | ~ 175,000 | ~ 5,000 | ~ 7,000 |
| 50 MHz | 24 | 54 | ~ 642,000 | ~ 871,000 | ~ 25,000 | ~ 34,000 |
| 200 MHz | 24 | 54 | ~ 3,148,000 | ~ 4,271,000 | ~ 122,000 | ~ 166,000 |

*\* Results were gathered by restarting each simulation and clearing the configuration bits, load value, and prescaler bits from the timer in between attempts.*

*\* The “~” signifies an approximate value. Even with the chosen methods, it was hard to get a consistent value. Therefore, an average of 5 attempts was taken for each result present in the table.*

From the data in the table above, we can deduce that per constant frame time, frequency determines how much time is spent servicing interrupts. Though the time spent within *SERVICE\_IRQ* is very trivial compared to the scale of instructions executed and memory accesses of *check\_switches* and *IDLE*, we can still say that as the effective frequency decreases, the more time will be spent servicing interrupts since the number of times an interrupt will be raised by the timer *T = 1 / f* which dictates an inverse relation to the effective frequency. In sum, we can say that we spend the most amount of time polling the switches followed by executing user code and lastly servicing interrupts.