**Summary**

The purpose of this project was to program a custom counter operation to the Simon board with two inputs, increment and decrement, and display the results using the led lights provided. The program is also required to generate a sound to the on board speaker whenever the counter “rolls over” (from 0 to F or F to 0).

To begin, we discovered that the counter variable had to be represented as a 4-bit number (0h-Fh) which initially provided an issue, the 8051 has no 4-bit memory addresses, only 8-bit and single bit addressable locations. To solve this we decided to execute all of our operations on the lower nibble of a memory location, originally we chose a bit addressable location (20h) but then settled on Register 5 (R5) in Bank 0. At the beginning of our program we initialize R5 to #00h, clear the Accumulator, and clear the Carry bit.

Our main “function” or routine is a large continuous loop which in order performs the following:

* Calls a RESET subroutine which conditionally resets the value of R5 to 0, and sounds the alarm whenever the counter rolls over.
* Moves the value of R5 into the accumulator and complements the bits of the accumulator. (**ACC** is bit addressable)
* The values of the lower 4 bits of the accumulator (E0h - E3h) are sent to corresponding LEDs which display the count. This let us to directly represent each bit of the count to a corresponding LED at all times. The accumulator was complemented because the LEDs are active low.
* Clears the carry bit which was used in moving the the **ACC** bits to the LEDs
* Jumps to the AUTOINC operation whenever Switch #4 is engaged. This is a special feature which counts up continuously as long as the button is held.
* Jumps to the DECREMENT operation whenever Switch #1 is engaged.
* Jumps to the INCREMENT operation whenever Switch #2 is engaged.
* Jumps to the CLEAR operation whenever Switch #3 is engaged. This is a special feature added feature which resets the value of R5 and calls the ALARM.
* Jumps to the BEEPCNT operation whenever Switch #5 is engaged. This is a special feature added that sounds the ALARM a given number of times equal to the decimal representation of count.
* Jumps back to the beginning of the loop.

To perform the INCREMENT and DECREMENT operations we moved the value of R5 into the accumulator and then used the **add** or **subb** commands to change the count by a value of #01h. We chose to use the **add** and **subb** rather than **INC** and **DEC** because we needed the operations to update the **CY** and **AC** flags of the Program Status Word, which we use in our RESET subroutine. The operation then moves the value of the accumulator back into R5 and executes a continuous loop keeping the program from jumping back to our main routine until the button press is released.

Determining how to sound the alarm whenever the count “rolled over” proved to be the most difficult task of the project. First to sound the alarm (or generate a noise from the speaker) we created a delay subroutine to generate a 1000 hz square wave (math included on last page). We send this square wave to the speaker pin to produce a sound. The given frequency however delivered a very fast time interval making it almost too fast to hear the alarm. To solve this we devised a loop within the ALARM subroutine to increase the duration of the alarm sound.

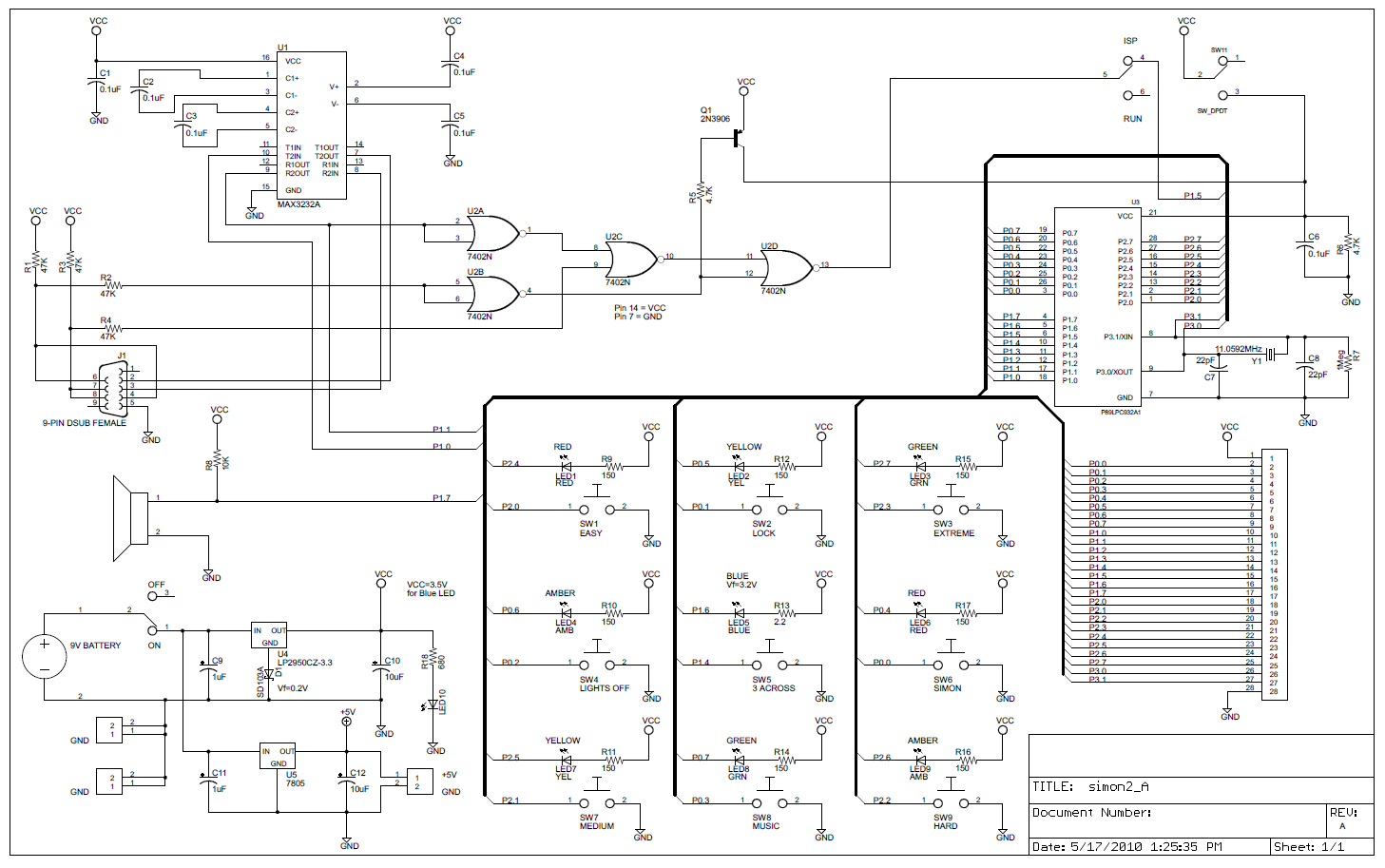
Once we discovered how to produce a sound upon the speaker, the difficult problem was calculating exactly when to call the ALARM routine, and understanding how to recognize that the count had “rolled over”. Since the Register we were using to store our count value into was an 8-bit register and our count was only a 4-bit value, we couldn’t use typical conditional jumps such as **jz** to achieve our goal. After many unsuccessful attempts and careful examination an idea was brought forth to check the status of the Carry bit and Auxiliary Carry bits after our Increment and Decrement operations. We discovered that by using the **add** and **subb** commands in our operations, the **AC** flag of the **PSW** would be set to a 1 whenever the lower nibble rolled over during addition, and that the **CY** flag of the **PSW** would be set to a 1 whenever the entire 8-bit word was rolled over during subtraction (as the **CY** acts as a borrow). Using these values we created a RESET subroutine which would conditionally check these two values after each operation, determine which bit, if any, was set, then jump to a corresponding block of code which reset the value of R5 to #00h when incrementing and #0Fh when decrementing and cleared the **CY** and **AC** bits (**PSW.7 and PSW.6**).

If neither of these bits were set, the RESET subroutine would just return back to where it was called. Using this RESET subroutine we ensure that the 8-bit value of the count never get out of the range of the 4-bit boundary (00h-0Fh). To aid in the testing of our ALARM routine we also set LED3 to turn on whilst the routine was running. We decided to leave this function in the program for added effect.

Similar to our strategy in Project 1, we mapped the pins corresponding to the LEDs and Switches as well as the Speaker, to constant values for easy of readability and programmability, during the project. We also set the utilized ports (P0M1, P1M1, and P2M1) to a value of 00h, to enable them for both reading and writing.

**Simon Board**

|  |  |  |  |
| --- | --- | --- | --- |
| **Port** | **Function** | **Port** | **Function** |
| P2.4 | SW1 - Decrement | P2.0 | LED1 |
| P0.1 | SW2 - Increment | P0.5 | LED2 |
| P2.3 | SW3 - Clear | P2.7 | LED3 |
| P0.2 | SW4 - Auto Inc | P0.6 | LED4 |
| P1.4 | SW5 - Beep CNT | P1.6 | LED5 |
| P1.7 | SPKR - Speaker | P0.4 | LED6 |
|  |  | P2.5 | LED7 |
|  |  | P0.7 | LED8 |
|  |  | P2.6 | LED9 |

****

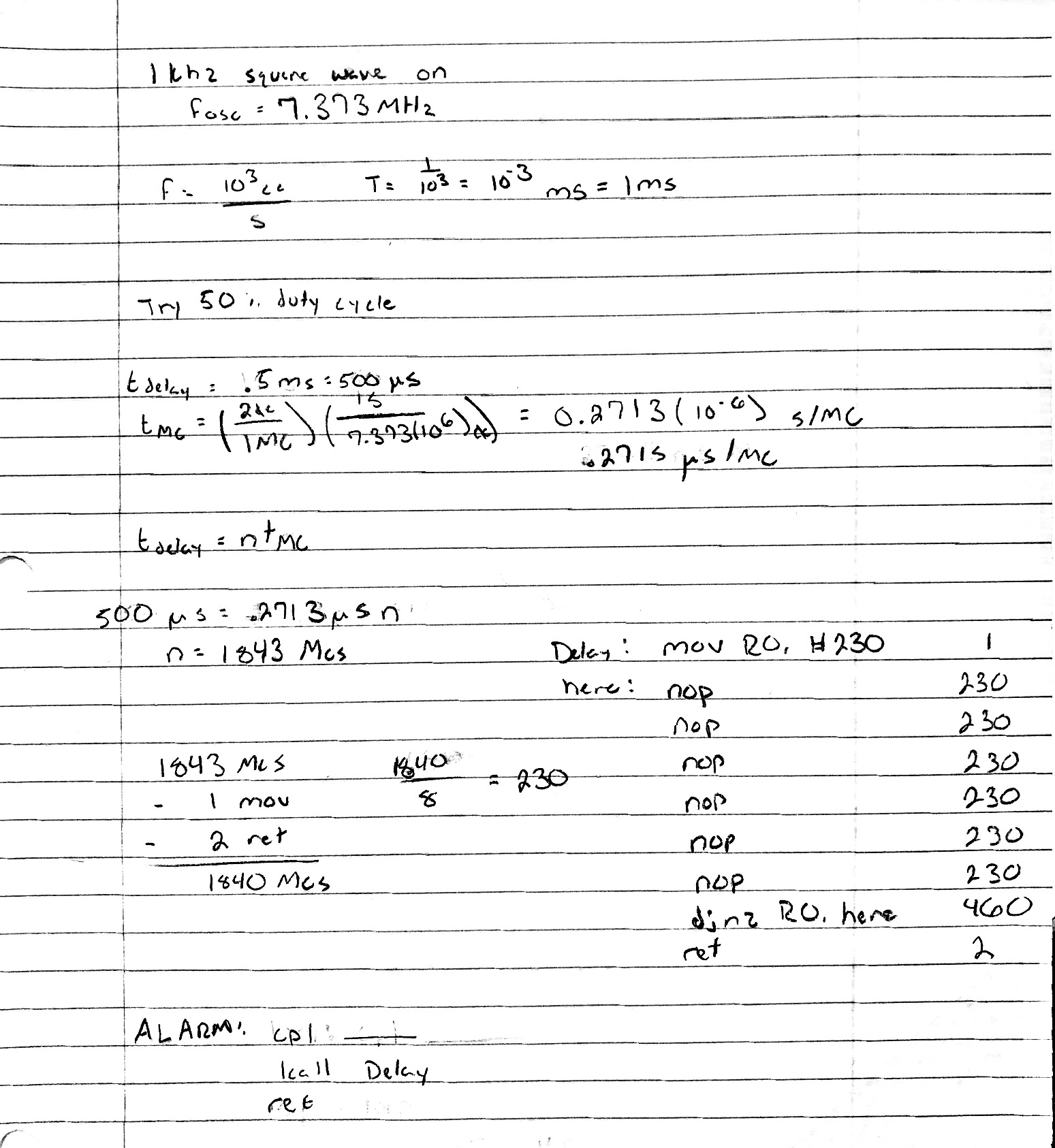
**Project Code(Well documented code is worth more, as is code that makes**

**appropriate use of segments, variables, labels, etc.)**

**Work Effort and Future Work**

Work effort was divided up equally among the group. All work with the exception of special feature implementation was completed during team meetings in which all members were present and contributed. Ideas were presented verbally from each member during meetings where one person was in charge of writing and compiling the code. This tended to be Jacques a majority of the time.

As a group we found it difficult to consistently apply principles of software engineering. Organizing the code in a more functional manner would have made it more manageable from a functional hierarchy perspective. Unfortunately due to the limited facilities available to us by the MCS-51 instruction set achieving the desired level of abstraction is essentially impossible, as Katherine correctly pointed out on at least two separate occasions, our code looks ugly. Jump statements seem to be closely related to Goto statements in higher level languages, which are frowned upon due to the messy unreadable code they inherently generate, often referred to as “spaghetti-code”. Regrettably, only one of our team members is actually old enough to remember when such adjectives were more than antiquated anachronisms. On a more serious note, this lack of abstraction cause us to discover that we had to carefully account for the offsets between jumps from routines, as on more than one occasion we ran into “error A51: TARGET OUT OF RANGE”.

**Sound Wave Math**