COE 4DS4 – Lab #3 Report

Group Number: 33
Moshiur Howlader / 001316948 / howlam@mcmaster.ca
Ryan Ganeshan / 001322407 / ganesr3@mcmaster.ca
January 31th, 2018

Exercise 1: This exercise involved extending experiment 1b to implement the specifications. A struct was made to represent the elevator and its control parameters. This struct was volatile so that polling-type logic can be used. We created a 12-index array to house the requests. The requests were taken from the switches by populating the array with a '1' at the given floor on either a falling/ rising edge. This was done in the ISR. The push buttons were taken care of in a similar way. Flags were raised and dropped depending on the edge within ISR of PB_3. We implemented relatively simple logic to service the floor requests. Priority was given according to the current direction of the elevator. To arrive at a floor, a custom counter component was reset and held until an IRQ was raised for its expire. This was done X times, where x corresponds to the delta between the current floor and the next floor. Overall, the logic was easy to implement, but the qsys setup was difficult to attain.

Exercise 2: This exercise was completed by modifying in-lab experiment 2. First, a flag was created to detect when the caps lock key was pressed. This flag was raised when we detected a make code corresponding to the caps-lock key. Then a separate case statement was implemented when this flag was high. The character key multiple-print-while-held issue was solved by implementing a 49-index array. This essentially creates a flag for each key to detect whether the key has been "freshly pressed". The flag is asserted when the interrupt sends a make code. It is cleared when the interrupt sends a break code. The PS2_translate code is run before the flags are decided. Thus, when a make code is detected the next time the interrupt is raised, the ISR will not run because a break code has not been detected for that key yet, meaning it has not yet been let go. This array implementation allows for multiple key presses. At the same time. The TA's have told us that due to hardware delays, the multiple keys cannot be pressed at the EXACT same time.

Exercise 3: i) We have implemented the MD5 algorithm using Brad Conte's C implementation found here: http://bradconte.com/md5_c. For the 3a setup, the run time average in clock cycle (excluding the outlier) was 119093. 3b setup gave an average runtime of 33763cc. 3c setup gave an average runtime of 33793.83 cc. The result for 3a makes sense, since the setup for Nios/e does not have neither instruction cache, branch prediction, nor the barrel shifter to reduce many clock cycles worth of computation MD5 requires. What does make sense in terms of performance for 3b and 3c compared to 3a is the 70% improvement in performance from the extra hardware these systems provide. Data cache certainly speed up many of the repetitive matrix computations done in the MD5 but storing and giving quick access to these constant values. Branch predication circuitry should not affect the performance runtime of MD5 since there is not many branching (if-else) code in the algorithm. Despite the fact that the expected runtime of Nios/f (3c) should have been significantly lower than Nios/s (3b), the two performance ended up being the same. Nios/f should have been significantly faster in performance because the use of barrel shifters should have saved many clock cycles worth of computation compared to Nios (3b). MD5 has many bit shifting calculations.
ii) The requested function prototype was implemented as: void findStringEqual33(char *str, int index, int len). The function uses recursion to iterate through each bit of the string input one at a time to produce every possible combination that was specified (a to z, and 0 to 1) for a given length len of input string. Our hit rate for N = 4 was 31. The hash array at hash[15] and hash[0] was checked to match our group number of 0x21 or 33 in decimal.