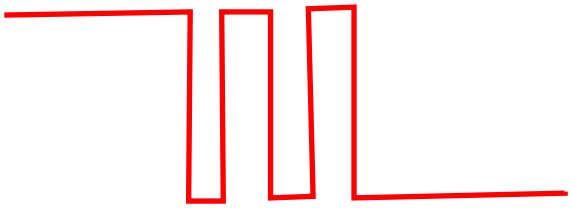
ME333 Introduction to Mechatronics
2022 Quiz 5: Chapter 5 & 6

Name Sean Morton

1. Draw the voltage that is generated by a “bouncey” push button, and describe a method for debouncing it using code in an interrupt.



A way to debounce it with code: check for a falling edge of the signal, then delay for 10ms. If the button is pressed after this delay period (i.e. V is LOW), then this was a button press, so we should carry out an ISR once. If the button is not pressed after this delay period, there was probably some switch bounce when we let go of the button, so exit the ISR and do nothing.

2. What are each of the IFSx, IECx, and IPCx registers used for?

IFSx: Interrupt Flag Status. Determines if the interrupt has been triggered recently (0-1)
IECx: Interrupt Enable Control. Determines if we have enabled a given interrupt in our program (0-1)
IPCx: Interrupt Priority Control. Determines the priority and queueing of interrupts, whether or not one interrupt should interrupt another, and in what order they should be executed
- Priority: 3 bits in IPCx, range 0-7
- Subpriority: 2 bits in IPCx, range 0-3

3. What is “context save and restore”, and how do you avoid using it?

"Context save and restore": when you jump from main() to your ISR due to an interrupt, you need a way to remember the values of variables you were using in main(). This method saves all the variables in main() in a separate part of memory, jumps to the ISR, and reloads the values of the variables when we return to main()

This method takes time, so we can avoid using it by using the Shadow Register Set. When we run main(), variables are stored in 2 places, with the SRS being the backup location. When we jump to an ISR, we don't have to save & restore variable values in main() because they're stored in the SRS.

4. The following methods of calculating velocity produce about the same results. What is the advantage of the second method, and how could you use the .dis file to prove it?

```
...
//method 1
float distance_F=3.0, time_F=2.0, speed_F; //units are in m and s
speed_F = distance_F / time_F; //units are m/s
//method 2
int distance_I=3, time_I=2, speed_I; //units are in m and s
speed_I = (distance_I*1000) / time_I; //units are mm/s
...
```

The advantage of the second method: float math is really slow because it requires special subroutines to do addition, subtraction, multiplication, and division. Int math is much faster. The second method avoids having to have a float result (1500 mm/s as opposed to 1.5 m/s) so the result probably won't be calculated using any float subroutines.

We can prove this in the .dis file using 2 things: # of lines of Assembly, and jumps to other subroutines through command <jal> or similar. Fewer # of lines and a lack of float math subroutines should make our program have less commands, and run faster.