CMPSC 153A: LAB 2A

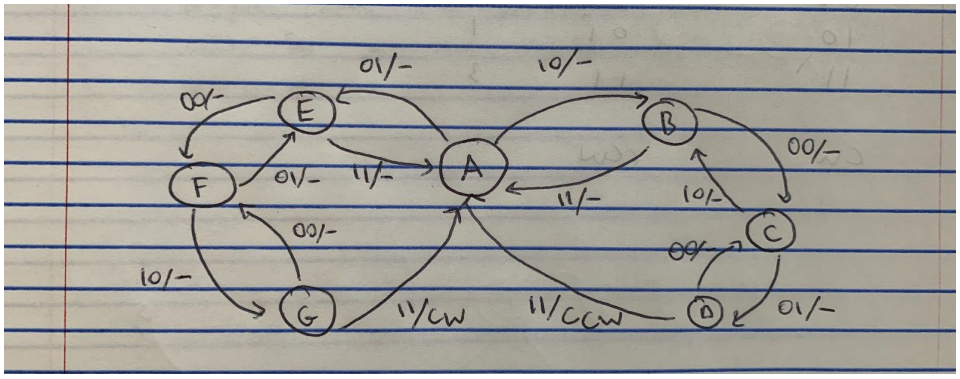
Alex Shortt

Lab 2A explored new peripherals - RGB LEDs and a rotary encoder -and utilized finite state machines to control them.

Methodology

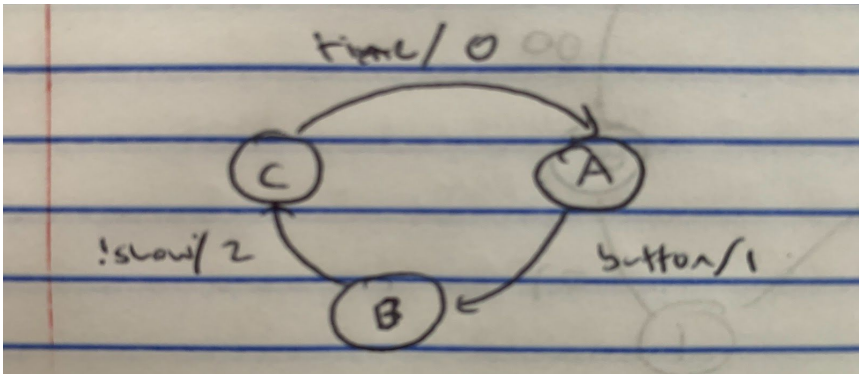
The specs of the project were simple: control the position of the led along the row of leds with a rotary encoder, with the click feature of the encoder to toggle the display. The led was to move left on a counter-clockwise turn and right on a clockwise turn, but while the display was off it was not supposed to move. Alongside this, we were to blink the RGB LEDs consistently to indicate that the program was running successfully.

The encoder functionality was implemented using, you guessed it, interrupts. The issue came that the interrupt fired on every state change for the encoder, and since it's a physical device prone to imperfection, there would be bouncing values as the switch was turned. As such, the solution for cleanly capturing full rotations without spazzing out was to use an FSM. I opted to pass in the values of both pins as the input to the FSM with freedom to move linearly through the states of the full rotation. The state machine looks as follows:



As shown above, a full CCW rotation goes { 11, 10, 00, 01, 11 }, and a CW rotation goes { 11, 01, 00, 10, 11 }. It took me a while to realize that the transitions to previous states are crucial for the debouncing so that the final { 11 } input never gets out of sync with the actual state of the rotation. The outputs would get stored in a global variable for the grand loop to consume and act upon.

On the other hand, the button on the rotary encoder was much easier to implement. Basically, the FSM would start at state A, and if a button push was detected it would move to state B. From there, the grand loop would transition the FSM to state C and perform the button functionality of toggling the led display. Finally, for the sake of debouncing, after half a second had passed the grand loop would transition the FSM back to A, where it would be ready for another button press.



Results

In the end the program ran very smoothly. The debouncing method worked wonders for both the rotary encoder and the button, to where even me going out of my way to try to break it came up short. One part that I specifically enjoyed coding was the FSM. I needed a modular way to store the state's reactions to inputs and its outputs without losing my mind while trying to keep track of all the 1's and 0's. I ended up defining letters to match my diagram to make it a lot more readable, and stored the transitions in a 2D array.

```
int TURN_FSM[7][4] = {
// next state for 00, 01, 10, 11
          {A, B, E, A}, // A
          {C, B, B, A}, // B
          {C, B, D, C}, // C
          {C, D, D, A}, // D
          {F, E, E, A}, // E
          {F, G, E, F}, // F
          {F, G, G, A}  // G
};
```

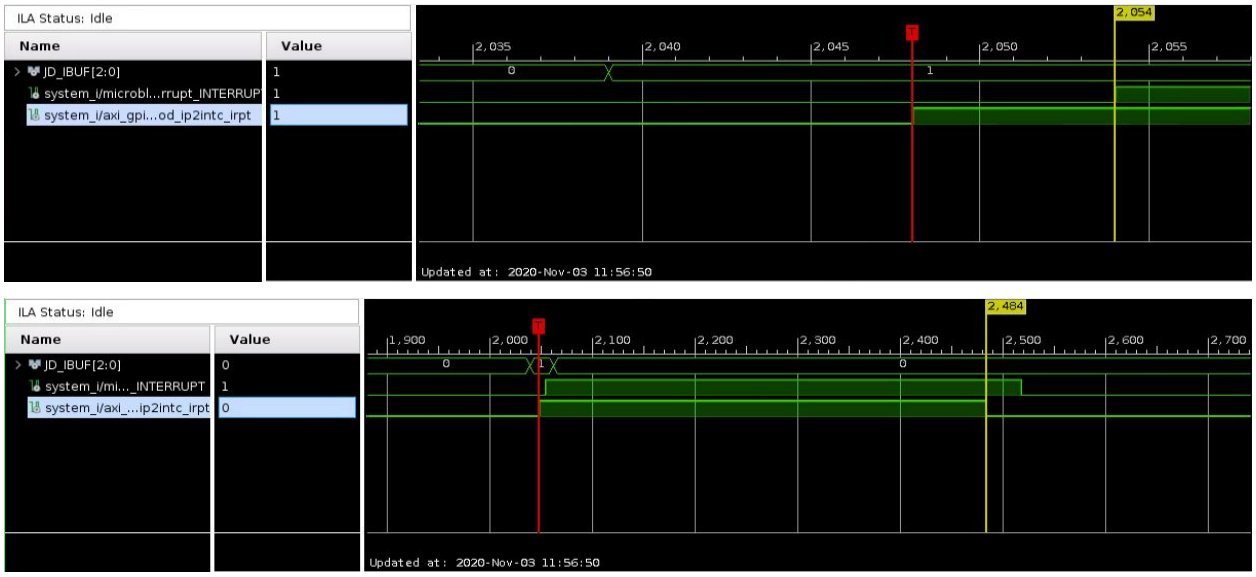
This array stored the next state, which when a new input was received would be read for determining the next state to travel to given the input and the current state.

```
int move_turn_fsm(int cur_state_index, int input) {
      int next_state = TURN_FSM[cur_state_index][input];
      ...
```

```
}
```

This implementation was simple and easy to configure, which I did - my first FSM was way wrong!

Following the lab instructions, I ran the debugger while running my program to catch exactly when the events related to the rotary encoder were running. The results are below, with the cycles between the input change to the GPIO Interrupt coming in at 9 cycles and the cycles between the GPIO Interrupt and the processor interrupt coming in at 6 cycles. I couldn't figure out how to track the address of the instruction register to see when it changed from the processor interrupt to 0x00000010, but considering that the encoder interrupt handler took 430 cycles, I would assume that's around how long it took.



Overall, it was clear especially from the debug view why bit 30 of the MSR exists: it keeps track of when an interrupt is running so the program knows (a) to stop any other interrupts from occurring and (b) how to proceed after the interrupt has been handled. It's clear from the diagram above that after handling the interrupt on my code it needs to handle the interrupt on the processor. How amazing.