Lab 2A: Implementing a Finite State Machine-Based Rotary Encoder Interface

Oviya Seeniraj

2. Purpose and Expected Goals

The purpose of this lab was to design and implement an interface for a rotary encoder using a finite state machine (FSM) using debouncing and directional control for 16 LEDs. We designed a method to toggle LED status with an encoder push button. The design goals were to develop an FSM for accurate and stable encoder twist detection with debouncing, use GPIO and interrupts to control LED movements, create a push-button toggle for LEDs, enable system status indication using an RGB LED, and ensure reliable performance with rapid encoder inputs (implement debouncing).

3. Methodology

a. Overview of Design Tasks

1. Hardware Setup: Modify the Vivado project from Lab 1B to add two GPIO devices: a 6-bit output GPIO for the RGB LEDs, and a 3-bit input GPIO for the encoder (channels A, B, and push button).
2. Interrupt Configuration: Enable interrupts on the encoder GPIO and connect it to the system's interrupt controller.
3. Develop FSMs for the rotary encoder to:
    - Debounce twists using state transitions based on observed sequence from quadrature encoder.
    - Debounce the push button with timer-based delays.
4. Create routines for LED control and rotary encoder state management in Vitis
    - LED movement functions (led_left(), led_right(), led_toggle())
    - FSM-based interrupt handlers for the encoder twist and push button
5. System Indicator: Implemented a loop with an RGB LED to indicate system status

b. Assumptions Required for Design Analysis and Procedures

- Default Encoder Signal State: A and B are high when idle; the push button defaults to low.
- Sequence Recognition: The encoder outputs specific sequences for clockwise and counterclockwise rotations (e.g., 11 → 01 → 00 → 10).
- Debouncing Characteristics: Bouncing patterns were assumed as minor state regressions that the FSM could handle by returning to previous states until stabilization.
- Cycle Time for Debounce: Assumed sufficient delay between input state changes to eliminate errant transitions due to bounce effects.

c. Observations from the Design Tasks

- The encoder FSM stabilized effectively even with quick twists.
- Timer-based debouncing for the push button was responsive with minimal lag.
- Rapid turns of the encoder were successfully managed with correct LED movements.

d. Plan for Design Testing

1. GPIO Signal Monitoring: Use Vivado hardware debugger to track signals from encoder & LED control.
2. Test rotary encoder FSM by observing LED wrapping behavior amd measuring LED response delay for clockwise and counterclockwise rotations.
3. System Stress Testing: Tested for stability with rapid encoder inputs to verify debounce resilience.

4. Results

<u>a. Metric Results</u>

- Goal AchievementAll goals were met, including LED movement, wrapping, and push-button toggle functionality.
- Performance Metrics
    - LED updates maintained within an acceptable delay range (<1ms).
    - Encoder responsiveness remained reliable even at fast twist rates.

<u>b. Limitations of the Design</u>

- Timer Granularity: System response depended on timer precision, potentially affecting high-speed toggles.
- Interrupt Overload Risk: Rapid consecutive encoder signals could risk interrupt flooding, although this was not observed in testing.

<u>c. Design Test Results for Assumption Deviations</u>

- Minor deviations from expected encoder twist sequence patterns were handled by the FSM, as bounces led the FSM to temporary states but did not disrupt the main sequence.
- Push button debounce timing proved consistent with assumptions, validating timer-based debounce as effective.

<u>d. Road Blocks and Issues</u>

- Initial issues arose with GPIO configuration in Vivado, particularly with configuring the Concat module and interrupt enablement.
- Debugging multiple GPIO interrupt sources presented challenges; careful signal tracing was required to differentiate twist and push events.

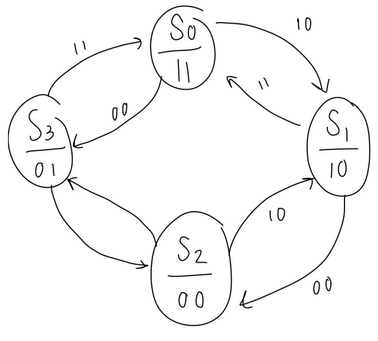
<u>e. Issues and Sources of Errors</u>

- Signal misinterpretation occurred in the FSM during early testing due to insufficient bounce handling. This was resolved by refining the debounce FSM transitions to filter repetitive inputs.
- Interrupt timing discrepancies were occasionally observed in the Vivado debugger due to cache invalidation issues.

5. Report Questions

<u>1. Explain how you decode which direction the rotary encoder is twisting in. Also explain how debouncing for the twists of the encoder was done. Show bubble charts (states, inputs and transitions) for the finite state machines in your design.</u>

We decode the encoder's rotation by analyzing the changes in the state of the two quadrature signals, A and B, which indicate the direction of movement. The bubble chart is to the right.



Nested behavior for each state determined whether we were in a clockwise or counterclockwise direction so that at S1 or S3 respectively we would move right (CCW) or left (CW) by tracking a global variable for how far

CW/CCW we were in clicks. I used this rather than making 7 states due to initial errors of states with the same values, and it was organizationally easy.

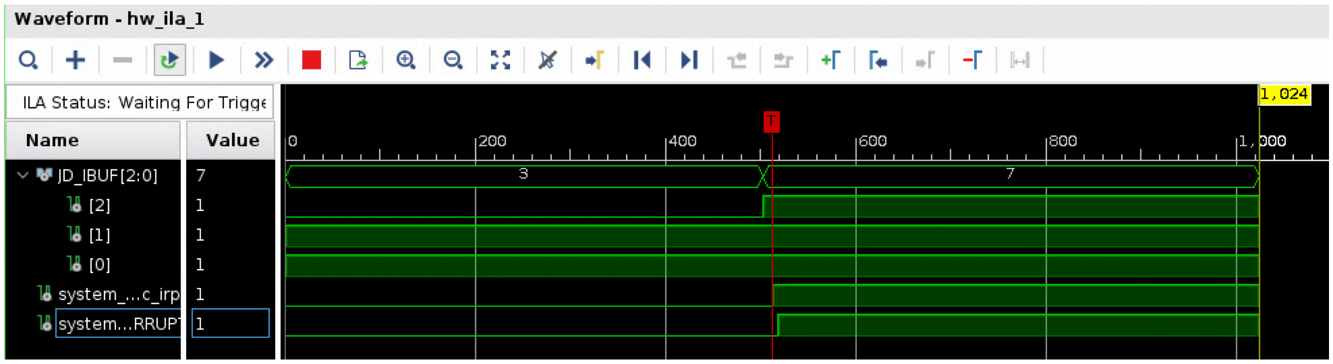
I implemented a debouncing mechanism with a delay after each twist by introducing a small delay (length chosen empirically) between each state change. After the initial detection of a twist, the system only registers a change if the state remains stable for a set period, ensuring that brief noise signals / erroneous inputs are ignored.

2. Explain how debouncing for the click of the encoder was done.

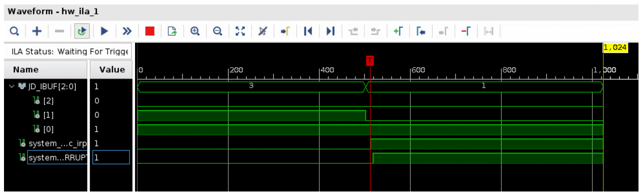
I implemented the encoder click debounce similarly to the twist. Upon detecting an encoder press, I introduced a delay to ensure that bouncing does not register as multiple clicks. The encoder click state must remain stable for a specific period before we recognize it as an actual press.

3. Include two screenshots from the hardware debugger.

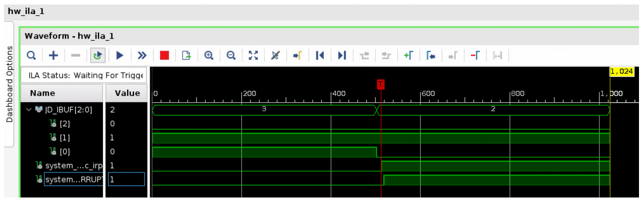
Interrupt from the push button and the GPIO signal:



Interrupt from the twist button and the GPIO signals (there are 2) for twist:

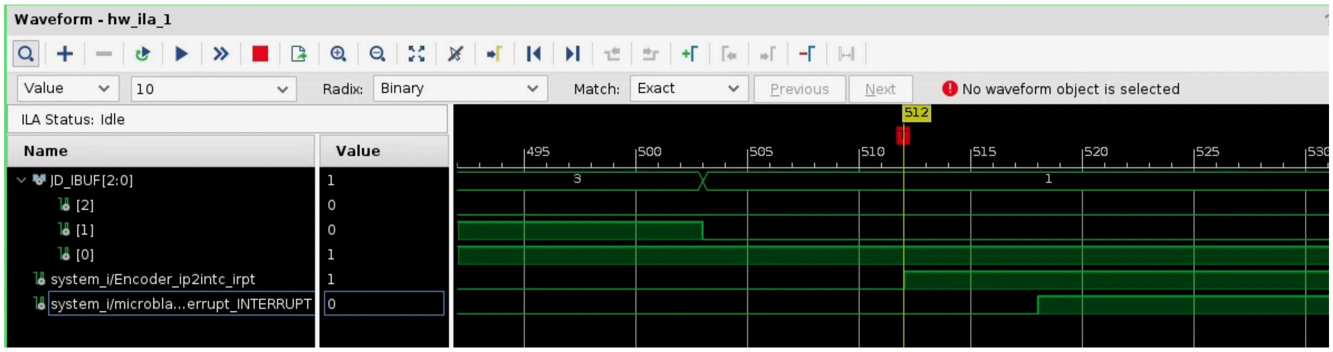


Clockwise twist



Counterclockwise twist

Zoomed view of cycles:



The captured waveform allowed us to measure the delays for each stage:

1. Input Change to GPIO Interrupt: 9 cycles
2. GPIO Interrupt to Processor Interrupt: 6 cycles

3. Processor Interrupt to Exception Handler (0x00000010): 4 cycles
4. Exception Handler to Encoder Interrupt Code Execution: 221 cycles

<u>4. What is the Microblaze MSR Register used for? What is the purpose of Bit 30 in the MSR?</u>

The MicroBlaze Machine Status Register (MSR) helps control interrupt handling and processor state. Bit 30 of the MSR is necessary for interrupt management - it is the Interrupt Enable (IE) bit. This bit allows the processor to respond to interrupt signals when set to 1. Clearing this bit (setting it to 0) disables interrupts, allowing the processor to complete critical operations without interruption. In this lab, by setting Bit 30, the processor can manage interrupts from the encoder, enabling timely responses to GPIO signals.