Lab Report: Implementing PWM with FPGA DE-10 Lite

Course: EECS3216 – Digital Systems Engineering

Lab 2 Report

Group Members:

- **Fares Trad** (Student ID: 217281379)
- **Kanwarjot Singh** (Student ID: 214284269)
- **Josh Keppo** (Student ID: 210971752)
- **Kyle Williamson (**Student ID: 218953901)

---

1. Introduction

This lab focuses on designing and implementing a sine wave inverter control system using an FPGA, employing digital PWM modulation through a Type-2 FSM. The process approximates sine waves by comparing carrier waves with precomputed values stored in a Look-Up Table, creating variable duty cycle outputs that simulate analog waveforms when filtered. The implementation incorporates a Mealy FSM for managing state transitions and timing, low-pass filtering to smooth PWM signals into AC waveforms, and real-time frequency control via push buttons with debouncing logic. The lab develops practical skills essential for power electronics applications such as AC motor control and power conversion systems.

---

2. Objective

Lab Tasks

1. **Generating a PWM Signal**

- Understand PWM as a technique to control average voltage by varying duty cycle
- Use the DE10-Lite's 50 MHz clock frequency
- Calculate counter maximum value (COUNTER_MAX = 49999) for 1 kHz PWM
- Implement duty cycle control by comparing counter value with threshold
- Generate and verify a 50% duty cycle signal on D0 using oscilloscope

2. **Modulating the PWM Signal to Simulate a Sine Wave**

    - Vary PWM duty cycle over time to simulate a sine wave
    - Implement a sine lookup table (LUT) with precomputed values
    - Create a phase counter to step through the LUT at desired frequency
    - Update PWM signal every period to modulate duty cycle
    - Confirm generation of an accurate sine wave approximation

3. **Adjusting Sine Wave Frequency via DE10-Lite Push Buttons**

    - Implement frequency control using FPGA push buttons
    - Allow dynamic frequency adjustment within predefined range
    - Modify PWM generator counter limit to change output waveform period
    - Add debouncing logic to prevent false triggers from mechanical switch bouncing
    - Integrate frequency adjustment into Mealy FSM for smooth transitions
    - Verify functionality using oscilloscope/logic analyzer
    - Use LED indicators as visual feedback for current frequency

---

3. PWM Design and Implementation

3.1 FSM Characteristics

This uses a type-II FSM following a **Mealy machine** model, meaning that the outputs depend on both the **state** and the **inputs**. The design incorporates:

- **PWM Counter:** Counts from 0 to COUNTER_MAX (49999) to create 1 kHz PWM signal
- **Phase Counter:** Tracks position in sine LUT to determine current duty cycle
- **Frequency Control State:** Modulation frequency changed by +/- 100 Hz per user button press (KEY[0] to decrease or KEY[1] to increase) between 100 Hz and 10 kHz.
- **A state transition back to S_RESET using SW[0].**
- **D0:** Fixed 50% duty cycle PWM signal
- **D1:** Sine-modulated PWM signa**l**
- **LED Indicators:** Visual feedback for current frequency setting
- **LUT-Based Approach:** Uses pre-calculated sine wave values stored in memory
- **Debouncing Logic:** Required for reliable button input processing
- **Synchronous Design:** All state transitions occur on clock edges

---

3.2 PWM System States

The FSM consists of three states:

State Name	Description
S_RESET	Initializes the entire system to known starting values. When triggered by either power-up or pressing the reset switch, this state zeroes the counter and phase index, turns off all outputs and LEDs, and calculates initial duty cycle values. This preparation ensures the sine wave generation begins from a consistent, well-defined state before transitioning to the counting state.
S_COUNT	Handles the actual generation of PWM signals by incrementing a counter 50 million times per second and comparing it against threshold values. For each counter value, the system sets the fixed reference output HIGH when below its 50% threshold and sets the modulated output HIGH when below its sine-adjusted threshold. This state continues until either a complete PWM cycle finishes or a button press is detected, at which point the system needs to update its parameters.

S_UPDATE	Advances the sine wave generation and processes any frequency changes. Here, the system calculates the next sine value from the lookup table, updates the modulated duty cycle threshold accordingly, and handles any pending frequency adjustments from button presses. This state ensures smooth progression through the sine pattern and provides the mechanism for frequency control before resetting the counter and returning to the counting state to begin the next PWM cycle.

State Transition Diagram:
A hand-drawn FSM diagram is attached (as per lab requirements).

3.3 PWM SystemVerilog Code

The FSM is implemented in **SystemVerilog** with a modular approach. The **main FSM module** (`pwm_modulated_sine`) includes:

- **State transitions using `always_comb` logic.**
- **State updates using `always_ff` triggered by the clock (`MAX10_CLK1_50`) or user input (`SW[0]`).**
- **Button debouncing logic for KEY[0] and KEY[1] using a counter.**
- **Uses multiple latching events to keep synchronization, inputs/registered are not change until a clock cycle happens.**

State Encoding

```
typedef enum logic [1:0] {
     S_RESET,
     S_COUNT,
     S_UPDATE
} state_t;

state_t current_state, next_state;
```

State Transition logic in `always_comb`

```
always_comb begin
```

```
     new_freq = freq;
     if (pending_key0 && (freq < MAX_FREQ))
          new_freq = freq + STEP_FREQ;
     else if (pending_key1 && (freq > MIN_FREQ))
          new_freq = freq - STEP_FREQ;
     if ((pending_key0 || pending_key1) || (counter ==
     counter_max))
          next_state = S_UPDATE;
     else
          next_state = S_COUNT;
end
```

State Transition logic in `always_ff`

```
always_ff @(posedge MAX10_CLK1_50) begin
     if (rst_sync) begin
          counter <= 0;
          phase <= 0;
          freq <= BASE_FREQ;
          d1_compare <= ((CLOCK_FREQ / BASE_FREQ)) >> 1;
          scaled <= 0;
          ARDUINO_IO[0] <= 0;
          ARDUINO_IO[1] <= 0;
          LEDR <= 10'b0;
     end
     else begin
          case (current_state)
               S_RESET: begin
                    counter <= 0;
                    phase <= 0;
                    d1_compare <= ((CLOCK_FREQ / freq)) >> 1;
                    ARDUINO_IO[0] <= 0;
                    ARDUINO_IO[1] <= 0;
                    LEDR <= 10'b0;
```

```
                end
                S_COUNT: begin
                     if (counter < counter_max)
                     counter <= counter + 1;
                     ARDUINO_IO[0] <= (counter < d0_compare);
                     ARDUINO_IO[1] <= (counter < d1_compare);
                     LEDR[0] <= (counter < d1_compare);
                     LEDR[9:1] <= 0;
                end
                S_UPDATE: begin
                     freq <= new_freq;
                     scaled <= amplitude * sine_lut[phase];
                     d1_compare <= (((CLOCK_FREQ / new_freq)) >>
                     1) + (scaled >>> 15);
                     phase <= (phase == SAMPLE_COUNT - 1) ? 0 :
                     phase + 1;
                     counter <= 0;
                     ARDUINO_IO[0] <= 0;
                     ARDUINO_IO[1] <= 0;
                     LEDR[0] <= 0;
                end
           endcase
     end
end
```

---

4. Debouncing and Synchronizing the Circuit

Mechanical buttons produce noise (bouncing), which can cause unintended state transitions. To counteract this, a **debounce circuit** is implemented using a counter. The circuit is also setup to keep the inputs synchronized with the system clock.

```
module button_debounce (
     input logic clk,
     input logic rst,
```

```
     input logic button,
     output logic pulse
);

     logic button_sync1, button_sync2;
     always_ff @(posedge clk or posedge rst) begin
          if (rst) begin
               button_sync1 <= 1'b0;
               button_sync2 <= 1'b0;
          end
          else begin
               button_sync1 <= button;
               button_sync2 <= button_sync1;
          end
     end

     localparam COUNTER_MAX = 20'hFFFFF;
     logic [19:0] counter;
     logic stable_button;
     always_ff @(posedge clk or posedge rst) begin
          if (rst) begin
               counter <= 0;
               stable_button <= button_sync2;
          end
          else begin
               if (button_sync2 != stable_button)
                    counter <= 0;
               else if (counter < COUNTER_MAX)
                    counter <= counter + 1;
               if (counter == COUNTER_MAX)
                    stable_button <= button_sync2;
          end
```

---

5. Testing and Verification

5.1 Testbench (tb_pwm_modulated_sine)

A comprehensive testbench was developed to verify the functionality and robustness of the PWM modulated sine wave generator. The testbench simulates all required behaviors and edge cases to ensure the design meets specifications under various conditions.

The testbench instantiates the `pwm_modulated_sine` module as the Device Under Test (DUT) and monitors both outputs and internal signals, including direct access to the FSM states for complete verification.

FSM State Transition Testing

The testbench explicitly monitors the `current_state` and `next_state` signals, verifying proper transitions between states:

```
// Monitor state transitions
wait(current_state == dut.S_COUNT);

$display("Current state: S_COUNT");
// Press a key to trigger state transition to UPDATE
key[0] = 0;
#100;

// Verify transition to UPDATE statewait(current_state ==
dut.S_UPDATE);
$display("State transitioned to: S_UPDATE");
```

These tests confirm that button presses properly trigger transitions from S_COUNT to S_UPDATE, and that normal operation returns to S_COUNT after parameter updates.

Frequency Limit Verification

To ensure frequency limits are enforced, the testbench attempts to exceed both the upper and lower bounds:

```
// Test upper frequency limit (10kHz)
repeat(100) begin  // Try to go beyond max
     KEY[0] = 0;
     #500000;
     KEY[0] = 1;
     #500000;
end
assert(dut.freq <= 10000) else $error("Frequency exceeds
MAX_FREQ");
// Test lower frequency limit (100Hz)
repeat(100) begin  // Try to go below min
     KEY[1] = 0;
     #1000000;
     KEY[1] = 1;
     #1000000;
end
assert(dut.freq >= 100) else $error("Frequency below MIN_FREQ");
```

These tests confirm that the frequency remains constrained between 100 Hz and 10 kHz regardless of how many times the buttons are pressed.

Sine Wave Generation Testing

The testbench monitors the Arduino output pins to verify both the fixed and modulated PWM signals:

```
ARDUINO_IO[0] produces a consistent 50% duty cycle reference
signal
ARDUINO_IO[1] produces a sine-modulated PWM signal with varying
duty cycle
```

By observing these signals over multiple PWM cycles, the testbench confirms proper sine wave generation through duty cycle modulation.

Frequency Adjustment Verification

To verify the 100 Hz frequency steps, the testbench systematically presses each button and monitors the frequency register:

```
// Test frequency increase (press KEY[0])
repeat(10) begin
     KEY[0] = 0;
     #1000000;
     KEY[0] = 1;
     #1000000;
     $display("Current frequency after increase: %d Hz",
     dut.freq);
end
```

These tests confirm that each KEY[0] press increases frequency by exactly 100 Hz and each KEY[1] press decreases it by 100 Hz, with appropriate bounds checking.

Reset Functionality Testing

The testbench verifies proper reset operation by asserting SW[0] during normal operation:

```
// Test reset during operation
SW[0] = 0;  // Assert reset
#100;
assert(current_state == dut.S_RESET) else $error("Reset did not
transition to S_RESET state");
```

This confirms that the system immediately enters the S_RESET state when SW[0] is asserted low and properly restarts after reset is deasserted.

Through comprehensive simulation with this testbench, all design requirements were successfully verified, giving high confidence in the correctness of the PWM modulated sine wave generator implementation.

---

6. FPGA Implementation

Hardware Testing (DE10-Lite)

1. **Uploaded bitstream** to FPGA.
2. **Verified PWM output of various frequencies and state transitions.**
3. **Confirmed stable button response with debouncing.**
4. **Checked signal output with Oscilloscope.**

---

7. Challenges and Solutions

During the implementation of the PWM modulated sine wave generator, several technical challenges were encountered. This section outlines these challenges and the engineering solutions developed to address them.

Challenge	Solution
Synchronization	Latch events to only be updated on next clock cycle.
Button Debouncing	Implements a counter-based button debouncing logic.
Verifying Correctness	Use oscilloscope to show correct behavior, seeing a simulated sine wave.

7.1 Synchronization

Challenge:
Ensuring proper timing between asynchronous button presses and the synchronous PWM generation was critical. Without proper synchronization, frequency changes could occur at unpredictable points within a PWM cycle, potentially causing glitches or distortion in the output waveform.

Solution:
The system implements an event latching mechanism where button presses are stored in pending registers (`pending_key0` and `pending_key1`) until they can be processed at the appropriate time:

```
// Latch a new event if detected
pending_key0 <= (current_state == S_UPDATE) ? 1'b0 :
(pending_key0 || key0_pulse);
pending_key1 <= (current_state == S_UPDATE) ? 1'b0 :
(pending_key1 || key1_pulse);
```

This approach ensures that frequency changes only occur during the UPDATE state, maintaining clean transitions between PWM cycles and preserving the integrity of the sine wave pattern.

7.2 Button Debouncing

Challenge:
Physical button presses produce multiple electrical transitions due to mechanical bouncing. Without debouncing, a single button press might register as multiple frequency changes, making precise frequency control impossible.

Solution:
A dedicated `button_debounce` module was implemented with three key components:

1. **Input Synchronization:** Raw button signals are synchronized to the clock domain using a two-flip-flop synchronizer.

2. **Counter-Based Filtering:** A counter tracks how long the button signal remains stable:

```
if (button_sync2 != stable_button)
  counter <= 0; // Restart counter when a change is detected
else if (counter < COUNTER_MAX)
  counter <= counter + 1;
```

3. **Edge Detection:** Once the signal stabilizes, a single-pulse generator creates exactly one pulse per button press:

```
pulse <= (~stable_button_prev) & stable_button;
```

This debouncing solution ensures reliable frequency control, with each physical button press generating exactly one 100Hz frequency change.

7.3 Verifying Correctness

Challenge:
Confirming that the PWM modulation correctly produces a sine wave is difficult through simulation alone. The sine wave pattern emerges only after the PWM signal is filtered, which typically happens externally to the FPGA.

Solution:
A comprehensive verification strategy was employed:

1. **Testbench Simulation:** Verifies the internal logic, state transitions, and duty cycle modulation patterns.

2. **Oscilloscope Measurements:** Direct measurement of PWM outputs allows visualization of:
    - Fixed 50% duty cycle on ARDUINO_IO[0] as reference
    - Varying duty cycle on ARDUINO_IO[1] showing sine modulation
    - Frequency changes in response to button presses

3. **External RC Filtering:** By connecting a simple RC low-pass filter to ARDUINO_IO[1], the resulting sine wave can be observed directly on an oscilloscope, confirming proper generation across the frequency range (100Hz to 10kHz).

This multi-level verification approach ensured that the PWM modulation correctly generated the intended sine wave output with adjustable frequency.

---

8. Conclusion

This project aimed to implement a PWM-modulated sine wave generator using SystemVerilog on an FPGA. The objectives included generating a sine wave with adjustable frequency, implementing a Type-II Mealy FSM for state control, using a Look-Up Table (LUT) for sine wave approximation, and enabling real-time frequency adjustments via push buttons. Despite challenges with synchronization, button debouncing, and timing, we achieved a functional system where the frequency could be adjusted in 100 Hz increments between 100 Hz and 10 kHz. The project highlighted the potential of FPGAs for mixed-signal applications, though PWM's limitations in perfectly

replicating analog waveforms became evident. Future improvements could include better filtering, more precise timing, and additional waveform types. Overall, the project offered valuable hands-on experience with digital design, even if the results didn't always match expectations.

---

9. References

- EECS3216 Lecture Notes
- EECS3216 Lab 2 Solution
- Quartus Prime Documentation
- SystemVerilog IEEE Standard