Sample Solution for Lab 2 - PWM Modulated Sine Wave in SystemVerilog

2025-02-17

1 Introduction

This lab focused on designing and implementing a sine wave inverter logic control unit using an FPGA, emphasizing digital PWM modulation with a Type-2 FSM. It provides hands-on experience in digitally generating sine waves, a critical skill for AC motor control, power conversion, and other real-world applications.

The core objective was to approximate a sine wave using PWM modulation. You generated PWM outputs by comparing a carrier wave with precomputed sine wave samples stored in a Look-Up Table (LUT). This method ensures duty cycle modulation, effectively synthesizing a feed for AC signal and down stream gate drivers.

The lab introduces key PWM concepts, demonstrating how duty cycle variation simulates analog waveforms. By carefully adjusting sine LUT values, the PWM signal closely replicates a pure sine wave. A central component of this implementation is a time-based Mealy FSM, which manages state transitions, counters, and LUT addressing, ensuring accurate PWM synchronization. Additionally, we explore low-pass filters for smoothing high-frequency PWM into a usable AC output.

To bridge theory with practice, we provided a Python simulation that visualizes fixed and modulated PWM signals, as well as low-pass filtering effects. This enhanced your understanding of sine wave synthesis via PWM. In the second part, we introduced real-time frequency control using FPGA push buttons, modifying PWM counter limits for dynamic adjustment. Debouncing logic ensures reliable operation, and the FSM smoothly transitions between frequencies. Below we present a SystemVerilog module that generates a PWM-based sine wave feed with adjustable frequency:

2 Sample Verilog Code

```
//============================================================================
// pwm_modulated_sine: PWM modulated sine generator with integrated Mealy FSM,
// synchronized reset from SW[0], and button debouncing for KEY[0] and KEY[1]
// ARDUINO PIN 0: fixed 50% duty cycle PWM
// ARDUINO PIN 1: Modulated PWM
//============================================================================
module pwm_modulated_sine (
    input  logic        MAX10_CLK1_50,    // 50MHz clock
    input  logic [1:0]  SW,               // Board switches (SW[0] used for reset)
    input  logic [1:0]  KEY,              // Push buttons for frequency control
    output logic        ARDUINO_IO[1:0],  // PWM outputs
    output logic [9:0]  LEDR              // LED dimming / status outputs
);

  //----------------------------------------------------------------------
  // Reset Generation: SW[0] is active low. We create an asynchronous
  // reset (rst_async) and then synchronize its deassertion with a two-FF
  // synchronizer to generate rst_sync.
  //----------------------------------------------------------------------
  logic rst_async;
```

```
assign rst_async = ~SW[0];  // Active when SW[0] is low

reg rst_ff1, rst_ff2;
always_ff @(posedge MAX10_CLK1_50 or posedge rst_async) begin
    if (rst_async) begin
        rst_ff1 <= 1'b1;
        rst_ff2 <= 1'b1;
    end
    else begin
        rst_ff1 <= 1'b0;
        rst_ff2 <= rst_ff1;
    end
end
assign rst_sync = rst_ff2;

//---------------------------------------------------------------------------
// Parameters
//---------------------------------------------------------------------------
localparam CLOCK_FREQ   = 50_000_000;
localparam BASE_FREQ    = 1000;   // Initial frequency = 1 kHz
localparam SAMPLE_COUNT = 200;    // 200-point sine LUT
localparam MIN_FREQ     = 100;    // Minimum frequency = 100 Hz
localparam MAX_FREQ     = 10000;  // Maximum frequency = 10 kHz
localparam STEP_FREQ    = 100;    // Step up or down frequency change 100 Hz

//---------------------------------------------------------------------------
// Debounced push-button pulses for frequency control.
// KEY[0] increases frequency; KEY[1] decreases frequency.
//---------------------------------------------------------------------------
logic key0_pulse, key1_pulse;
button_debounce deb0 (
    .clk(MAX10_CLK1_50),
    .rst(rst_sync),
    .button(KEY[0]),
    .pulse(key0_pulse)
);
button_debounce deb1 (
    .clk(MAX10_CLK1_50),
    .rst(rst_sync),
    .button(KEY[1]),
    .pulse(key1_pulse)
);

//---------------------------------------------------------------------------
// Frequency and PWM control registers and variables
//---------------------------------------------------------------------------
logic [31:0] freq;         // Current frequency in Hz
logic [31:0] counter;      // PWM period counter
logic [31:0] counter_max;  // Maximum counter value (PWM period)
logic [31:0] d0_compare;   // 50% duty cycle compare value
logic [31:0] d1_compare;   // Modulated duty cycle compare value
logic [15:0] phase;        // Phase index into sine LUT
logic signed [31:0] scaled;    // Scaled sine value (for modulation)
logic [15:0] amplitude;    // Amplitude for modulation




// python code to get the Sine LUT values below:
// import numpy as np
// SAMPLE_COUNT = 200
// n = np.arange(SAMPLE_COUNT)
// sine_values = np.sin(2 * np.pi * n / SAMPLE_COUNT)
// q15_lut = np.round(sine_values * 32767).astype(int)
// print("logic signed [15:0] sine_lut [0:SAMPLE_COUNT-1] = '{")
// print("    " + ",\n    ".join(
//     ", ".join(f"{val:6d}" for val in q15_lut[i:i+10])
//     for i in range(0, SAMPLE_COUNT, 10)
// ))
// print("};")

//------------------------------------------------------------------------
// Sine LUT (16 bit quantized format)
//------------------------------------------------------------------------
logic signed [15:0] sine_lut [0:SAMPLE_COUNT-1] = '{
         0,   1029,   2057,   3084,   4107,   5126,   6140,   7148,   8149,   9142,
    10126,  11099,  12062,  13013,  13952,  14876,  15786,  16680,  17557,  18418,
    19260,  20083,  20886,  21669,  22431,  23170,  23886,  24579,  25247,  25891,
    26509,  27101,  27666,  28204,  28714,  29196,  29648,  30072,  30466,  30830,
    31163,  31466,  31738,  31978,  32187,  32364,  32509,  32622,  32702,  32751,
    32767,  32751,  32702,  32622,  32509,  32364,  32187,  31978,  31738,  31466,
    31163,  30830,  30466,  30072,  29648,  29196,  28714,  28204,  27666,  27101,
    26509,  25891,  25247,  24579,  23886,  23170,  22431,  21669,  20886,  20083,
    19260,  18418,  17557,  16680,  15786,  14876,  13952,  13013,  12062,  11099,
    10126,   9142,   8149,   7148,   6140,   5126,   4107,   3084,   2057,   1029,
         0,  -1029,  -2057,  -3084,  -4107,  -5126,  -6140,  -7148,  -8149,  -9142,
   -10126, -11099, -12062, -13013, -13952, -14876, -15786, -16680, -17557, -18418,
   -19260, -20083, -20886, -21669, -22431, -23170, -23886, -24579, -25247, -25891,
   -26509, -27101, -27666, -28204, -28714, -29196, -29648, -30072, -30466, -30830,
   -31163, -31466, -31738, -31978, -32187, -32364, -32509, -32622, -32702, -32751,
   -32767, -32751, -32702, -32622, -32509, -32364, -32187, -31978, -31738, -31466,
   -31163, -30830, -30466, -30072, -29648, -29196, -28714, -28204, -27666, -27101,
   -26509, -25891, -25247, -24579, -23886, -23170, -22431, -21669, -20886, -20083,
   -19260, -18418, -17557, -16680, -15786, -14876, -13952, -13013, -12062, -11099,
   -10126,  -9142,  -8149,  -7148,  -6140,  -5126,  -4107,  -3084,  -2057,  -1029
};

//------------------------------------------------------------------------
// Compute counter_max, d0_compare, and amplitude.
//------------------------------------------------------------------------
always_ff @(posedge MAX10_CLK1_50) begin
    if (rst_sync) begin
        counter_max <= 0;
        d0_compare  <= 0;
        amplitude   <= 0;
    end
    else begin
        counter_max <= (CLOCK_FREQ / freq) - 1;
        d0_compare  <= ((CLOCK_FREQ / freq)) >> 1;
        amplitude   <= ((CLOCK_FREQ / freq)) >> 2;
    end
end

//------------------------------------------------------------------------
// Our pending key event latches for KEY[0] and KEY[1]
// These registers hold a key event until it is processed in S_UPDATE.
//------------------------------------------------------------------------
logic pending_key0, pending_key1;
always_ff @(posedge MAX10_CLK1_50 or posedge rst_sync) begin
    if (rst_sync) begin
```

```
            pending_key0 <= 1'b0;
            pending_key1 <= 1'b0;
        end
        else begin
            // 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);
        end
    end

    //------------------------------------------------------------------------
    // Mealy FSM for PWM generation, sine modulation, and frequency control.
    // We use combinational logic to immediately compute a new frequency if a
    // pending key event is present.
    //------------------------------------------------------------------------
    typedef enum logic [1:0] {
        S_RESET,   // Initialize registers
        S_COUNT,   // Generate PWM signals
        S_UPDATE   // Update modulation parameters (and frequency if needed)
    } state_t;

    state_t current_state, next_state;
    logic [31:0] new_freq;  // Combinationally computed new frequency

    // Combinational next-state and frequency-update logic (Mealy style)
    always_comb begin
        // Default: no frequency change.
        new_freq = freq;
        // Use latched key events to update new_freq immediately.
        if (pending_key0 && (freq < MAX_FREQ))
            new_freq = freq + STEP_FREQ;
        else if (pending_key1 && (freq > MIN_FREQ))
            new_freq = freq - STEP_FREQ;

        // Transition to S_UPDATE if a pending key event exists or the PWM period expires.
        if ((pending_key0 || pending_key1) || (counter == counter_max))
            next_state = S_UPDATE;
        else
            next_state = S_COUNT;
    end

    // FSM state update (synchronous reset)
    always_ff @(posedge MAX10_CLK1_50) begin
        if (rst_sync)
            current_state <= S_RESET;
        else
            current_state <= next_state;
    end

    // FSM outputs and actions
    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
                    // Immediately update the frequency using the combinationally
                    // computed new_freq.
                    freq <= new_freq;
                    // Recalculate modulation parameters using the (possibly) new frequency.
                    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

endmodule

//==============================================================================
// Button_debounce with a two-FF synchronizer, a counter-based filter, and a
//  rising-edge detector that generates a single pulse per button press.
//==============================================================================
module button_debounce (
    input  logic clk,      // Clock signal
    input  logic rst,      // Synchronous reset (e.g., rst_sync)
    input  logic button,   // Raw button input
    output logic pulse     // Single-cycle pulse on rising edge (debounced)
);

    // Step 1: Synchronize the raw button input to avoid metastability.
    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

    // Step 2: Debounce filter using a counter.
    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;  // Assume initial state is stable
        end
        else begin
            if (button_sync2 != stable_button)
                counter <= 0;  // Restart counter when a change is detected
            else if (counter < COUNTER_MAX)
                counter <= counter + 1;

            // Once stable for enough cycles, update the stable state.
            if (counter == COUNTER_MAX)
                stable_button <= button_sync2;
        end
    end

    // Step 3: Edge detector to generate a single pulse on the rising edge.
    logic stable_button_prev;
    always_ff @(posedge clk or posedge rst) begin
        if (rst) begin
            stable_button_prev <= 1'b0;
            pulse <= 1'b0;
        end
        else begin
            stable_button_prev <= stable_button;
            pulse <= (~stable_button_prev) & stable_button;
        end
    end

endmodule
```

3 Explanations and Discussions

As you see above, we used the following key parameters for our code:

- **CLOCK_FREQ:** 50 MHz clock frequency or our DE10-Lite Board
- **BASE_FREQ:** Initial operating frequency (1 kHz) for our PWMs.
- **SAMPLE_COUNT:** Number of points in the sine LUT (200 points).
- **Frequency Limits and Step:** Frequency can be adjusted between 100 Hz and 10 kHz in steps of 100 Hz.

We also used debouncing for our frequency control push button keys through our two instances of the `button_debounce` module ensure that only a single pulse is generated per physical press for KEY[0] (frequency increase) and KEY[1] (frequency decrease). We also latched the debounced pulses in registers (`pending_key0` and `pending_key1`) until they are processed by the combinational logic of our Mealy FSM. If a key press is detected, the frequency is updated immediately (increased or decreased) within the defined limits.

3.1 Why Reset Generation and Synchronization Necessary?

In our code, we used asynchronous reset through an external switch (SW[0], active low). This is inverted to generate an asynchronous reset signal (`rst_async`). When SW[0] is low, the reset is asserted.

We also created a synchronizer by using two flip-flops (`rst_ff1` and `rst_ff2`) to synchronize the deassertion of the reset with our 50 MHz clock. The final synchronized reset (`rst_sync`) is derived from the second flip-flop, ensuring safe operation in the synchronous domain.

As we covered in EEC3201, for our synchronous digital systems, asynchronous signals such as reset inputs must be properly synchronized to the system clock to prevent **metastability**. We can use **two-stage flip-flop synchronizer** to synchronize the deassertion of the reset signal (`rst_async`) with the 50 MHz clock. This is to avoid **Metastability** which occurs when an asynchronous signal transitions close to the clock edge, leading to an unpredictable or undefined state in flip-flops. A two-stage synchronizer ensures that the final derived reset signal (`rst_sync`) is stable and reliable.

3.1.1 Asynchronous Reset Signal Arrival

- The reset signal is **active-low** and inverted:
- $$\texttt{rst\_async} = \sim \texttt{SW[0]}$$
- When `SW[0]` is low, `rst_async` is asserted (high).
- When `SW[0]` is high, `rst_async` is deasserted (low).

3.1.2 Two Flip-Flop Synchronization Step

- **First Flip-Flop (`rst_ff1`)** captures `rst_async` on the next rising clock edge.
- **Second Flip-Flop (`rst_ff2`)** ensures a stable transition for `rst_sync`.

3.1.3 Synchronous Deassertion Step

- If `rst_async` is asserted (`1`), both `rst_ff1` and `rst_ff2` are set to `1`, forcing an immediate reset.
- If `rst_async` is deasserted (`0`), `rst_ff1` updates first, and then `rst_ff2` follows in the next clock cycle.

3.1.4 Why a Two-Flip-Flop Synchronizer?

- **Reduces Metastability:** The first flip-flop captures an unstable signal, while the second ensures it stabilizes.
- **Ensures Synchronous Reset Deassertion:** Reset is deasserted only on valid clock edges.
- **Prevents Glitches:** The flip-flops prevent transient pulses from causing unintended behavior.

3.2 Frequency Calculation and PWM Timing

- **PWM Period Calculation:** The period is derived by:

  $$\texttt{counter\_max} = \frac{\texttt{CLOCK\_FREQ}}{\texttt{freq}} - 1$$

  For instance, for a frequency of 1 kHz, the period is roughly 50 000 clock cycles of our 50 MHz clock.

- For testing, we have a **fixed 50% duty cycle**, which is obtained via a compare value (`d0_compare`):

  $$\texttt{d0\_compare} = \left(\frac{\texttt{CLOCK\_FREQ}}{\texttt{freq}}\right) \gg 1$$

  ensuring a constant 50% duty cycle on `ARDUINO_IO[0]`.

- **Amplitude Calculation:** The amplitude for modulation is set to one-quarter of the period:

$$\texttt{amplitude} = \left(\frac{\texttt{CLOCK\_FREQ}}{\texttt{freq}}\right) \gg 2$$

The amplitude calculation in a PWM sine wave generator determines the **modulation depth** of the sine wave, influencing how much the PWM duty cycle varies. This is essential for generating an accurate sine-modulated output signal. This determines how much the PWM duty cycle is adjusted according to the sine wave. Let's do an example:

- **CLOCK_FREQ** is the system clock frequency (50 MHz = 50,000,000 Hz).
- **freq** is the desired sine wave frequency (adjustable between 100 Hz and 10 kHz).
- The ratio $\frac{\text{CLOCK\_FREQ}}{\text{freq}}$ represents the number of **clock cycles per period** of the sine wave.
- for 1 kHz, each sine wave cycle takes **50,000 clock cycles**:

$$\frac{50,000,000}{1000} = 50,000$$

- As we know the right shift by 2 is equivalent to dividing by 4. And our **amplitude base value** will be **12,500**:

$$\left(\frac{\text{CLOCK\_FREQ}}{\text{freq}}\right) \div 4 \text{amplitude} = \frac{50,000}{4} = 12,500$$

Setting the amplitude to one-quarter of the period ensures:

- **Proportional duty cycle variation** based on the sine wave.
- **Controlled modulation depth**, avoiding extreme duty cycle values.
- **Eventual smooth waveform generation** with minimal distortion.

The amplitude value scales the sine wave lookup table (LUT):

- $$\text{scaled} = \text{amplitude} \times \text{sine\_lut[phase]}$$
- If sine_lut[phase] = 0.5 (50% of max amplitude),
- Then:

$$\text{scaled} = 12,500 \times 0.5 = 6,250$$

- This result is later used to modify the PWM duty cycle, producing a modulated waveform.

In our method, please note the effect of frequency changes on amplitude:

- $$\text{amplitude} \propto \frac{1}{\text{freq}}$$
- **Lower frequencies (e.g., 100 Hz)**: Higher amplitude, deeper modulation.
- **Higher frequencies (e.g., 10 kHz)**: Lower amplitude, less modulation.
- This maintains a relatively consistent waveform shape across different frequencies.

3.3 Sine LUT for Modulation

As shown in pre-lab python codes, we created:

- **Lookup Table (LUT):** A 200-element LUT (`sine_lut`). We store the quantized sine values in 16-bit format:

$$\texttt{sine\_lut[i]} = \texttt{round}\left(\sin\left(\frac{2\pi i}{\texttt{SAMPLE\_COUNT}}\right) \times 32767\right)$$

- The LUT provides discrete sine samples used to modulate the duty cycle of the PWM output (`ARDUINO_IO[1]`), such that after filtering, a sine waveform is obtained as demonstrated in the python code.

3.4 PWM Modulation and Duty Cycle Calculation

- **Fixed vs. Modulated PWM Outputs:**
  - `ARDUINO_IO[0]` is a fixed 50% duty cycle PWM.
  - `ARDUINO_IO[1]` is modulated using the sine LUT.
- **Modulated Compare Value:** In the S_UPDATE state, the modulated duty cycle is computed as:

$$\texttt{d1\_compare} = \left(\frac{\texttt{CLOCK\_FREQ}}{\texttt{new\_freq}}\right) \gg 1 + \left(\texttt{scaled} \ggg 15\right)$$

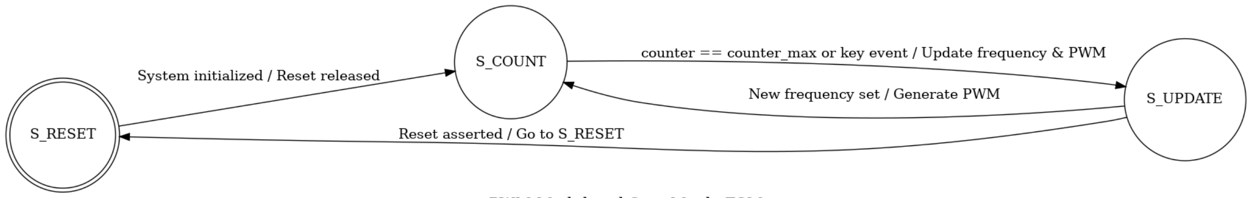
  where:

$$\texttt{scaled} = \texttt{amplitude} \times \texttt{sine\_lut[phase]}$$

  The arithmetic shift right with sign preservation (`>>> 15`) converts the Q15 product back to the same scale as the counter.

3.5 Mealy FSM Operation

We implemented our FSM in Mealy style, where state transitions and outputs depend on both the current state and input signals:


S_RESET
System initialized / Reset released
S_COUNT
counter == counter_max or key event / Update frequency & PWM
New frequency set / Generate PWM
S_UPDATE
Reset asserted / Go to S_RESET

PWM Modulated Sine Mealy FSM

- **S_RESET:** Initializes registers (counter, phase, frequency, and compare values) and ensures we start from a known state. We do this using dip switch 0.
- **S_COUNT:** Increments the counter until it reaches `counter_max`. PWM outputs are generated by comparing the counter with `d0_compare` and `d1_compare`.
- **S_UPDATE:** Updates the frequency (if a key event is pending), recalculates the modulation parameters, computes the new `d1_compare`, advances the sine LUT phase, and resets the counter.
- The FSM uses combinational logic to immediately calculate a new frequency based on key events:
  - KEY[0] increases the frequency if it is below the maximum limit.
  - KEY[1] decreases the frequency if it is above the minimum limit.
- A transition to S_UPDATE occurs when a key event is detected or the PWM period ends, ensuring prompt updates.