# Comparch Final Project

Michaela Fox and Maddy Fahey

## Introduction

This project extends the sine wave generator from Mini-Project 3 to create a multi-waveform generator system with real-time control over waveform type and frequency via an SPI interface. The design uses two iceBlinkPico FPGA boards, where one acts as an SPI master controller and the other as an SPI slave waveform generator. Waveforms supported include sine, triangle, and square waves. Frequency is controlled dynamically by sending a divider value over SPI. Output is converted to analog form using a 10-bit R-2R DAC connected to the GPIOs of the slave FPGA. SystemVerilog was used for all RTL modules and testbenches. The waveforms were observed via GTKWave and verified on an oscilloscope using an Analog Discovery 2.

---

## System Overview

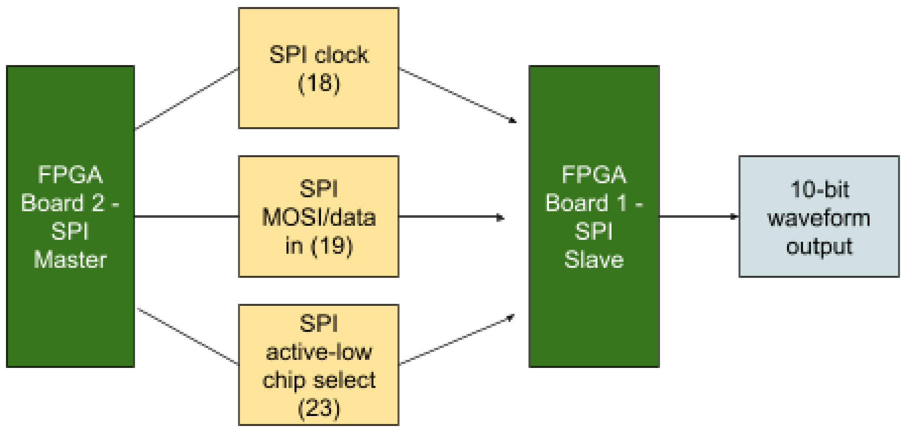
The complete system is split across two FPGA boards:

- **Board 1: SPI Slave + DAC Output:**
  - Receives an 18-bit SPI packet (16-bit frequency divider + 2-bit waveform select).
  - Generates the requested waveform at the desired frequency.
  - Drives a 10-bit DAC using GPIO pins for analog output.

- **Board 2: SPI Master:**
  - Waits for user input via a button press.
  - Cycles waveform type and sends new parameters to the slave board over SPI.

---

## Block Diagram



---

## Waveform Generator Design

The waveform generator on Board 1 produces sine, triangle, and square waves based on the SPI input.

- **Sine Wave:**
    - Uses a quarter-cycle lookup table (LUT) with 128 entries.
    - Based on quadrant selection, mirrors and/or inverts the LUT output.
    - Produces a 10-bit sine waveform centered at 512.

- **Triangle Wave:**
    - Linearly ramps up and down using the waveform address.
    - Output is scaled and centered to match DAC range (0–1023 or centered around 512 for analog symmetry).

- **Square Wave:**
    - Uses MSB of address to alternate between low and high output values.
    - 50% duty cycle with full amplitude swing (also centered for analog balance).

---

## SPI Communication Design

- **Master Design (Board 2):**
    - On each button press, the master:
        - Cycles through the waveform types (sine → square → triangle → sine).
        - Sends an 18-bit SPI packet:
            - Bits [17:2] = frequency divider
            - Bits [1:0] = waveform select
    - Implements a state machine (IDLE → ASSERT_CS → TRANSFER → DONE) to send SPI data synchronously.

- **Slave Design (Board 1):**
    - Samples SCLK rising edge to shift in SPI data (Mode 0).
    - When 18 bits are received:
        - Updates waveform selection register.
        - Updates clock divider register to set waveform output frequency.

---

## Frequency Control

The frequency of waveform output is controlled by a 16-bit divider value transmitted over SPI. The divider slows the address increment rate, effectively adjusting the output frequency.

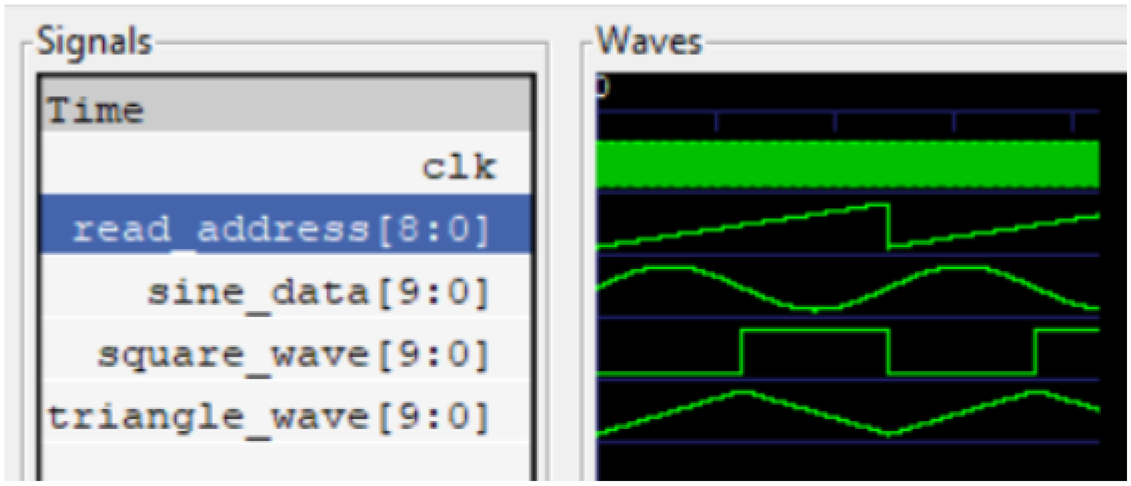
---

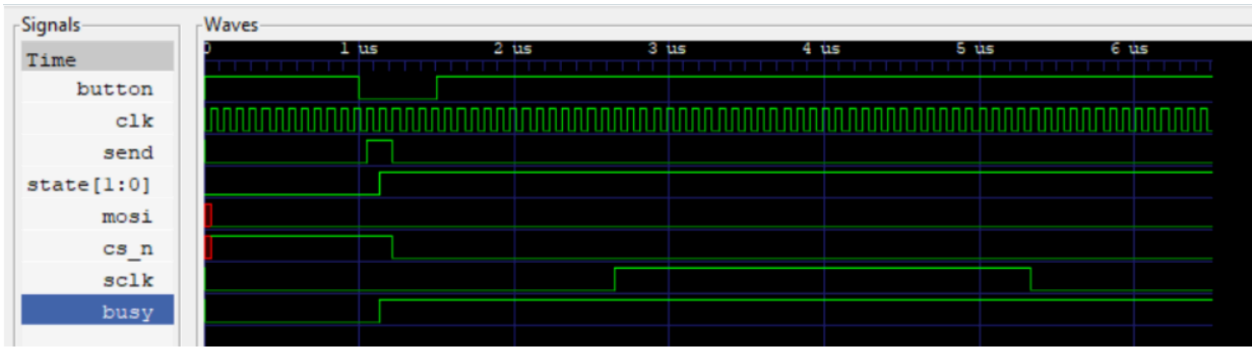
## Simulation Results

Simulation was performed using Icarus Verilog and GTKWave.

- Testbenches emulate SPI input and verify waveform selection and timing.
- Sample GTKWave output shows correct generation of all three waveforms.
- Output values align with expected sine, triangle, and square patterns.

**GTKWave Simulation Results:**





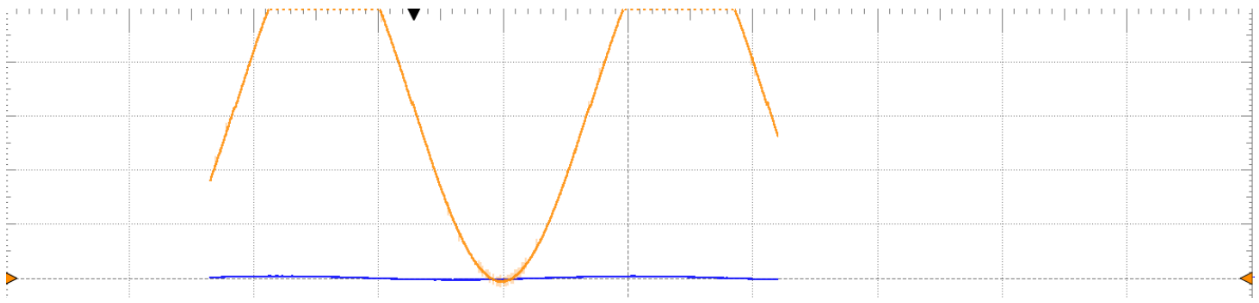
---

## Oscilloscope Results

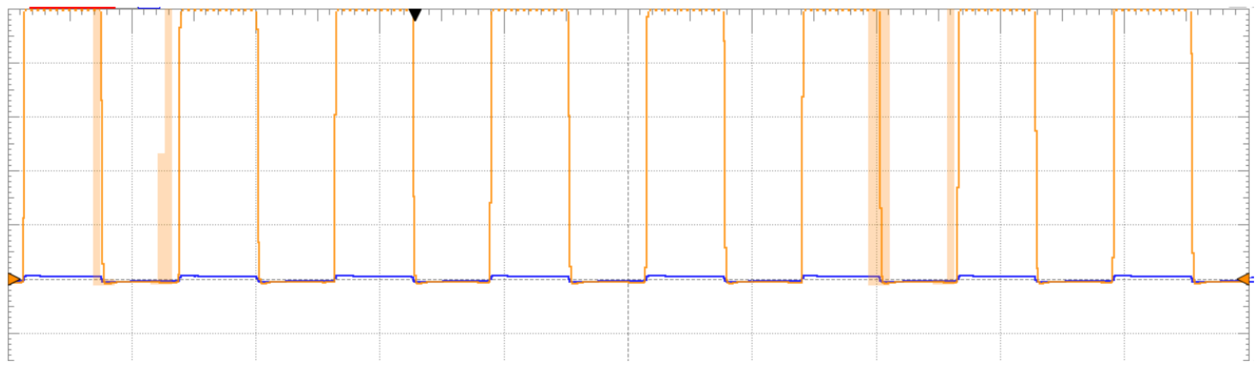
We used Analog Discovery 2 to capture real-world outputs from the R-2R DAC. Due to technical difficulties with the Waveforms software, we were unable to capture the full waveforms.

**Oscilloscope Waveforms:**

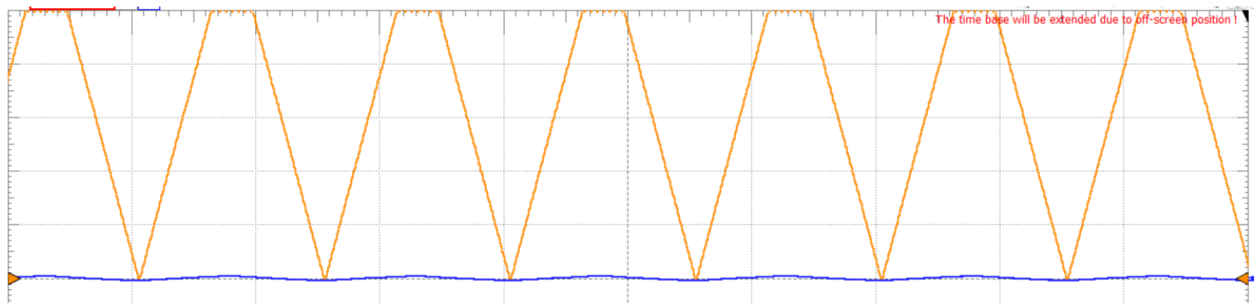
Sine:



Square:

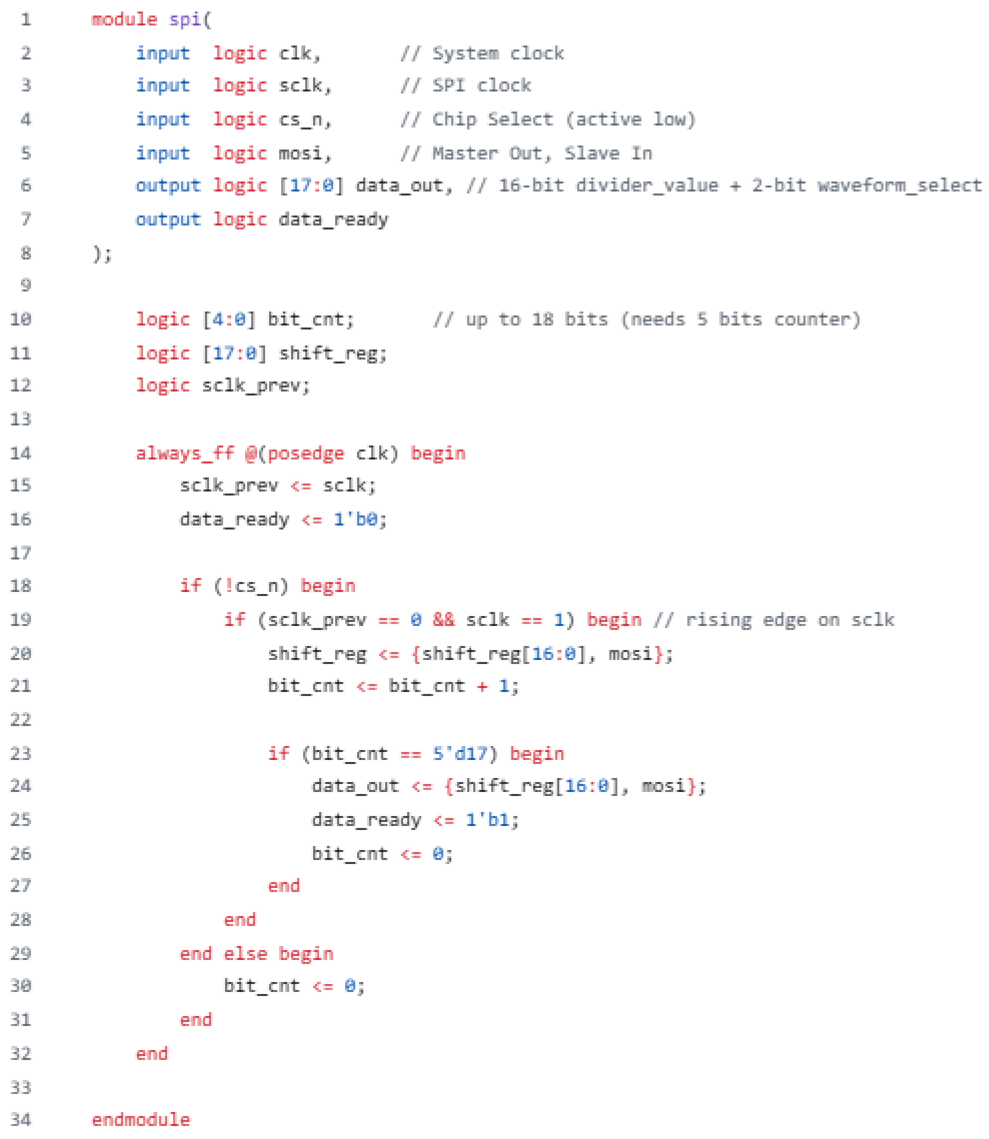


Triangle:



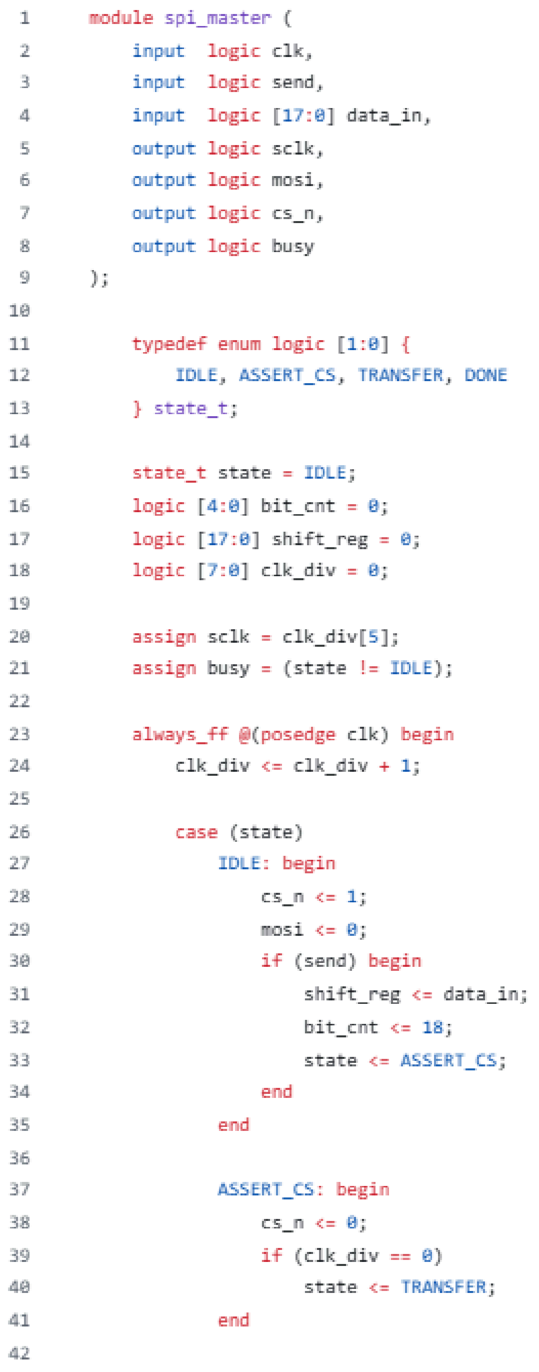
## Source Code

### **spi.sv:** - SPI data receiver, data latch



```
module spi(
    input  logic clk,       // System clock
    input  logic sclk,      // SPI clock
    input  logic cs_n,      // Chip Select (active low)
    input  logic mosi,      // Master Out, Slave In
    output logic [17:0] data_out, // 16-bit divider_value + 2-bit waveform_select
    output logic data_ready
);

    logic [4:0] bit_cnt;       // up to 18 bits (needs 5 bits counter)
    logic [17:0] shift_reg;
    logic sclk_prev;

    always_ff @(posedge clk) begin
        sclk_prev <= sclk;
        data_ready <= 1'b0;

        if (!cs_n) begin
            if (sclk_prev == 0 && sclk == 1) begin // rising edge on sclk
                shift_reg <= {shift_reg[16:0], mosi};
                bit_cnt <= bit_cnt + 1;

                if (bit_cnt == 5'd17) begin
                    data_out <= {shift_reg[16:0], mosi};
                    data_ready <= 1'b1;
                    bit_cnt <= 0;
                end
            end
        end else begin
            bit_cnt <= 0;
        end
    end

endmodule
```

**Spi_master.sv:** - SPI master FSM, bit shifter, CS control

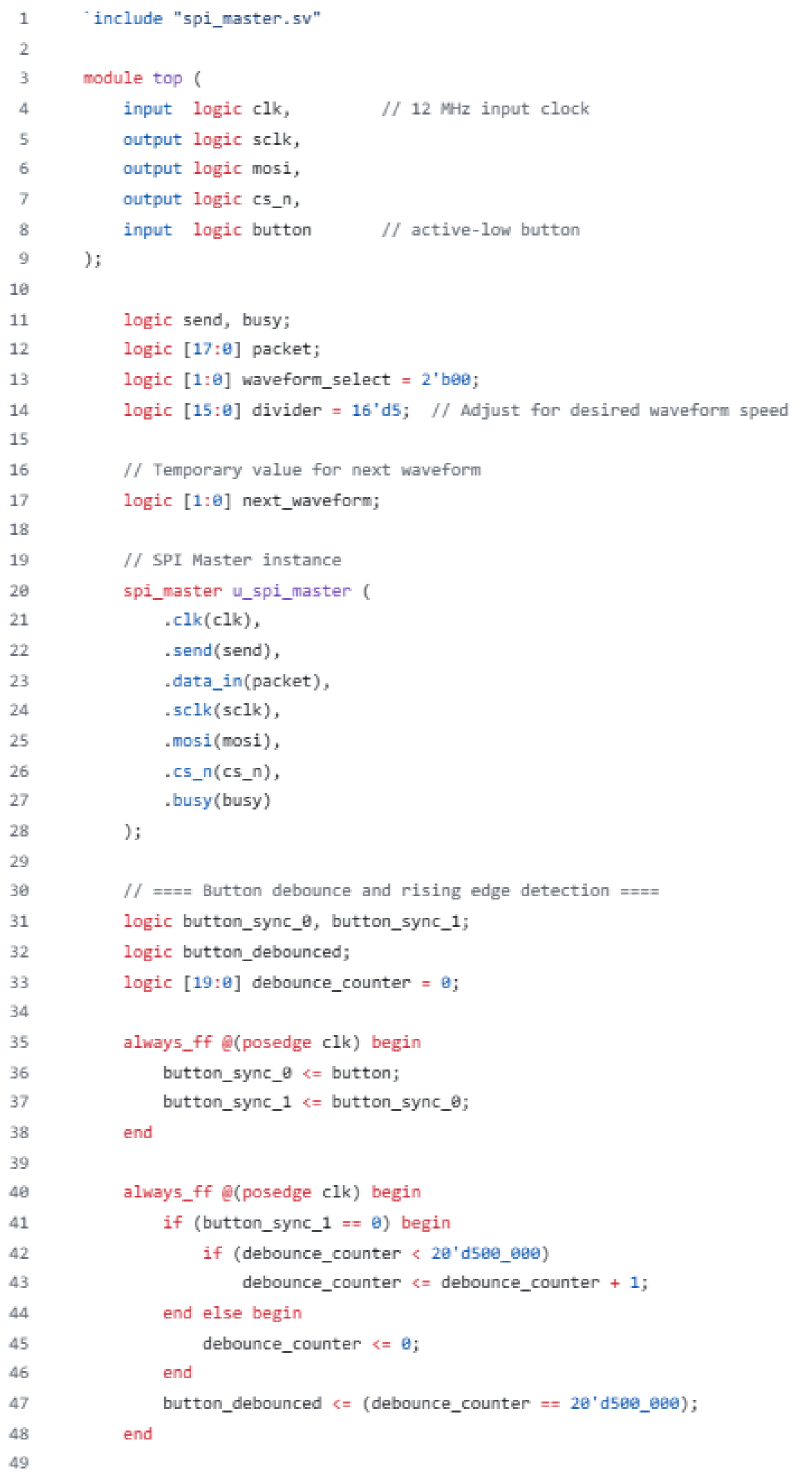


```systemverilog
module spi_master (
    input  logic clk,
    input  logic send,
    input  logic [17:0] data_in,
    output logic sclk,
    output logic mosi,
    output logic cs_n,
    output logic busy
);

    typedef enum logic [1:0] {
        IDLE, ASSERT_CS, TRANSFER, DONE
    } state_t;

    state_t state = IDLE;
    logic [4:0] bit_cnt = 0;
    logic [17:0] shift_reg = 0;
    logic [7:0] clk_div = 0;

    assign sclk = clk_div[5];
    assign busy = (state != IDLE);

    always_ff @(posedge clk) begin
        clk_div <= clk_div + 1;

        case (state)
            IDLE: begin
                cs_n <= 1;
                mosi <= 0;
                if (send) begin
                    shift_reg <= data_in;
                    bit_cnt <= 18;
                    state <= ASSERT_CS;
                end
            end

            ASSERT_CS: begin
                cs_n <= 0;
                if (clk_div == 0)
                    state <= TRANSFER;
            end

            TRANSFER: begin
                if (clk_div[5:0] == 6'b111111) begin
                    mosi <= shift_reg[17];
                    shift_reg <= {shift_reg[16:0], 1'b0};
                    bit_cnt <= bit_cnt - 1;
                    if (bit_cnt == 1)
                        state <= DONE;
                end
            end

            DONE: begin
                cs_n <= 1;
                if (clk_div == 0)
                    state <= IDLE;
            end
        endcase
    end

endmodule
```

**memory.sv:** - Sine LUT, quadrant-based full-wave reconstruction

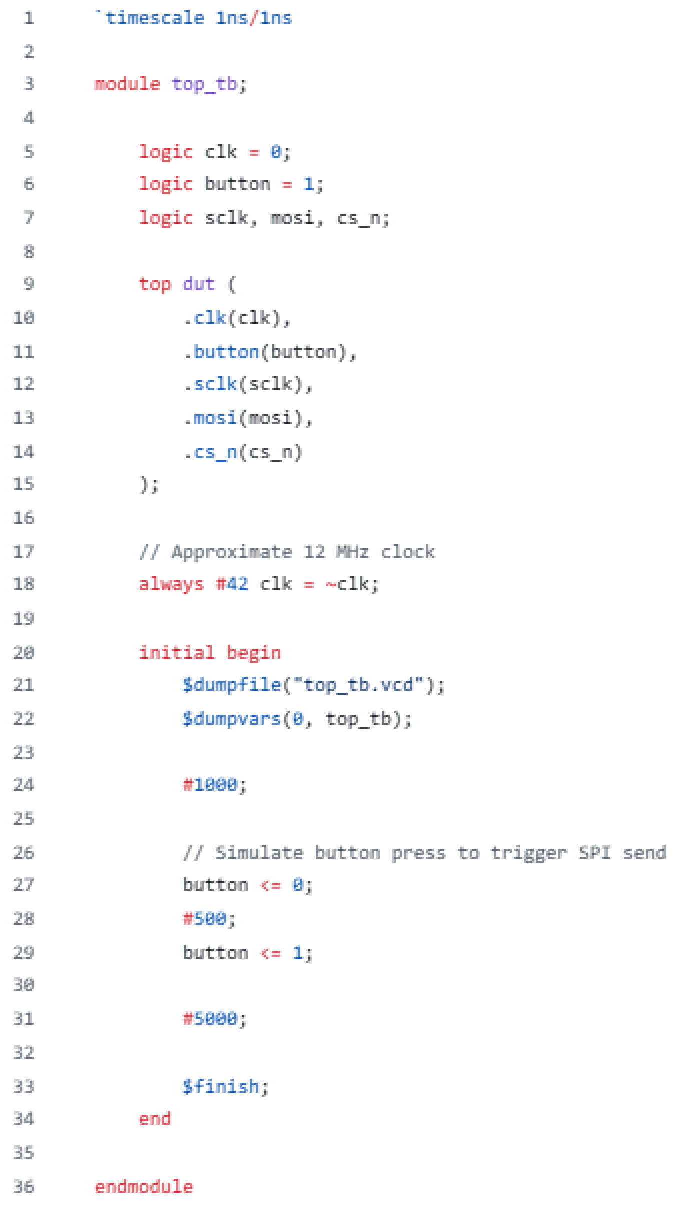
```
module memory #(
    parameter INIT_FILE = ""
)(
    input logic clk,
    input logic [8:0] read_address,
    output logic [9:0] read_data
);

    logic [8:0] quarter_cycle_memory [0:127]; // 9-bit amplitude (0-511)

    initial if (INIT_FILE) begin
        $readmemh(INIT_FILE, quarter_cycle_memory);
    end

    always_ff @(posedge clk) begin
        case (read_address[8:7])
            // Q1: 0-π/2 -> 512 to 1023 (512 + amplitude)
            2'b00: read_data <= 10'd512 + {1'b0, quarter_cycle_memory[read_address[6:0]]};

            // Q2: π/2-π -> 1023 to 512 (512 + mirrored amplitude)
            2'b01: read_data <= 10'd512 + {1'b0, quarter_cycle_memory[127 - read_address[6:0]]};

            // Q3: π-3π/2 -> 512 to 1 (512 - amplitude)
            2'b10: read_data <= 10'd512 - {1'b0, quarter_cycle_memory[read_address[6:0]]};

            // Q4: 3π/2-2π -> 1 to 512 (512 - mirrored amplitude)
            2'b11: read_data <= 10'd512 - {1'b0, quarter_cycle_memory[127 - read_address[6:0]]};
        endcase
    end

endmodule
```

**top.sv (Master): -** Divider logic, waveform selection, DAC driver



```
`include "spi_master.sv"

module top (
    input  logic clk,         // 12 MHz input clock
    output logic sclk,
    output logic mosi,
    output logic cs_n,
    input  logic button       // active-low button
);

    logic send, busy;
    logic [17:0] packet;
    logic [1:0] waveform_select = 2'b00;
    logic [15:0] divider = 16'd5;  // Adjust for desired waveform speed

    // Temporary value for next waveform
    logic [1:0] next_waveform;

    // SPI Master instance
    spi_master u_spi_master (
        .clk(clk),
        .send(send),
        .data_in(packet),
        .sclk(sclk),
        .mosi(mosi),
        .cs_n(cs_n),
        .busy(busy)
    );

    // ==== Button debounce and rising edge detection ====
    logic button_sync_0, button_sync_1;
    logic button_debounced;
    logic [19:0] debounce_counter = 0;

    always_ff @(posedge clk) begin
        button_sync_0 <= button;
        button_sync_1 <= button_sync_0;
    end

    always_ff @(posedge clk) begin
        if (button_sync_1 == 0) begin
            if (debounce_counter < 20'd500_000)
                debounce_counter <= debounce_counter + 1;
        end else begin
            debounce_counter <= 0;
        end
        button_debounced <= (debounce_counter == 20'd500_000);
    end

    // Edge detection
    logic button_debounced_prev;
    logic rising_edge;

    always_ff @(posedge clk) begin
        button_debounced_prev <= button_debounced;
        rising_edge <= (button_debounced && !button_debounced_prev);
    end

    // ==== Waveform selector and SPI send control ====
    logic send_request;

    always_ff @(posedge clk) begin
        if (rising_edge)
            send_request <= 1;
        else if (send)
            send_request <= 0;

        send <= (!busy && send_request);

        if (!busy && send_request) begin
            next_waveform = waveform_select + 1;
            waveform_select <= next_waveform;
            packet <= {divider, next_waveform};
        end
    end

endmodule
```

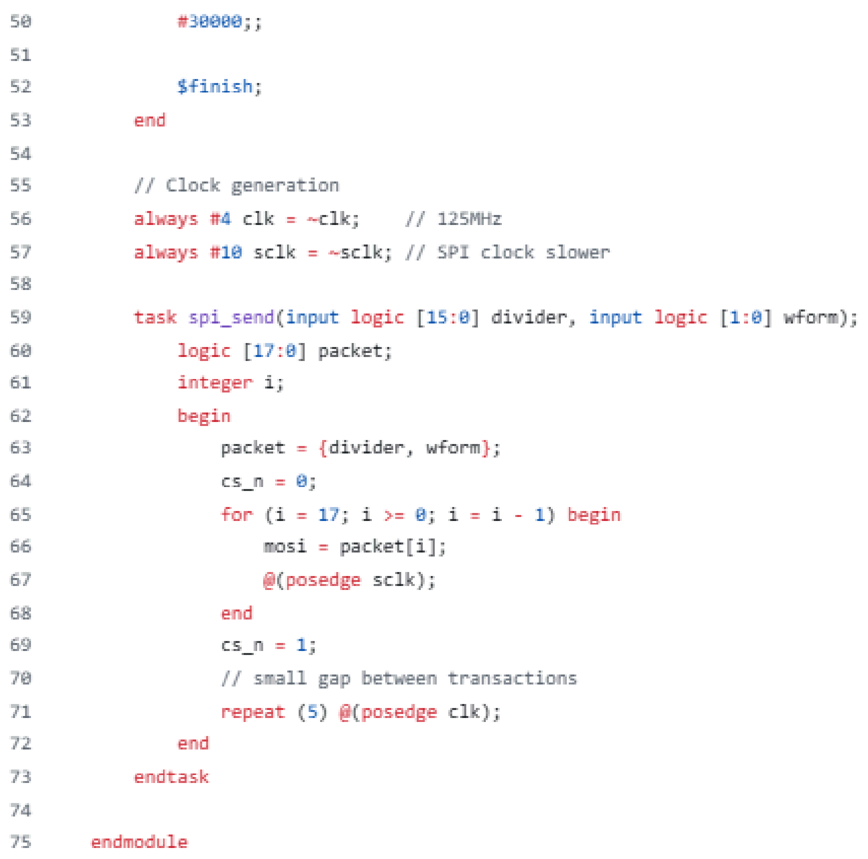
**Top_tb.sv:** - Verifies SPI master operation, button-triggered waveform cycling, and correct generation of SPI signals (SCLK, MOSI, CS_n)



```
`timescale 1ns/1ns

module top_tb;

    logic clk = 0;
    logic button = 1;
    logic sclk, mosi, cs_n;

    top dut (
        .clk(clk),
        .button(button),
        .sclk(sclk),
        .mosi(mosi),
        .cs_n(cs_n)
    );

    // Approximate 12 MHz clock
    always #42 clk = ~clk;

    initial begin
        $dumpfile("top_tb.vcd");
        $dumpvars(0, top_tb);

        #1000;

        // Simulate button press to trigger SPI send
        button <= 0;
        #500;
        button <= 1;

        #5000;

        $finish;
    end

endmodule
```

**Sine_tb.sv: -** Verifies SPI slave reception, waveform selection logic, and correct analog output generation for sine, triangle, and square waves.

```
`timescale 10ns/10ns

module sine_tb;

    logic clk = 0;
    logic sclk = 0;
    logic cs_n = 1;
    logic mosi;

    logic _9b, _6a, _4a, _2a, _0a, _5a, _3b, _49a, _45a, _48b;

    // Instantiate top DUT
    top dut (
        .clk(clk),
        .sclk(sclk),
        .cs_n(cs_n),
        .mosi(mosi),
        ._9b(_9b),
        ._6a(_6a),
        ._4a(_4a),
        ._2a(_2a),
        ._0a(_0a),
        ._5a(_5a),
        ._3b(_3b),
        ._49a(_49a),
        ._45a(_45a),
        ._48b(_48b)
    );

    initial begin
        $dumpfile("sine.vcd");
        $dumpvars(0, sine_tb);

        // Wait a little before sending SPI
        #1000;

        // SPI send: Divider 500, waveform select 10 (triangle wave)
        spi_send(16'd500, 2'b10);

        #5000;

        // SPI send: Divider 1000, waveform select 00 (sine wave)
        spi_send(16'd1000, 2'b00);

        #5000;

        // SPI send: Divider 250, waveform select 01 (square wave)
        spi_send(16'd250, 2'b01);

```


```
        #30000;;

        $finish;
    end

    // Clock generation
    always #4 clk = ~clk;    // 125MHz
    always #10 sclk = ~sclk; // SPI clock slower

    task spi_send(input logic [15:0] divider, input logic [1:0] wform);
        logic [17:0] packet;
        integer i;
        begin
            packet = {divider, wform};
            cs_n = 0;
            for (i = 17; i >= 0; i = i - 1) begin
                mosi = packet[i];
                @(posedge sclk);
            end
            cs_n = 1;
            // small gap between transactions
            repeat (5) @(posedge clk);
        end
    endtask

endmodule
```

## Demo Video

https://github.com/CompArchMiniProject4/iceBlinkPico/blob/main/examples/final/20250430_115411.mp4