FPGA-Based Ultrasonic Radar System with Real-Time VGA Display

Ada Zağyapan

Bilkent University

EEE102-2: Digital Logic Design

Cem Tekin

May 16, 2025

Introduction and Objective

Object detection and distance measurement play a crucial role in numerous real-world applications including autonomous navigation, collision avoidance, and robotics. Traditional radar and LiDAR systems often rely on complex analog hardware or costly sensors, whereas ultrasonic sensing offers a low-cost, compact alternative. This project aims to implement a real-time ultrasonic radar capable of scanning its environment, measuring distance via two ultrasonic sensors, and visualizing results on a VGA display. The design integrates servo-driven scanning, dual-sensor comparison, and lookup tables for sine and cosine computations in order to provide a digital approach to an ultrasonic radar system.

Design Methodology

Prior to any coding, research was conducted on each hardware component. The HC-SR04 ultrasonic sensors' timing characteristics, the servo motor's pulse-width requirements, and the piezo buzzer's drive thresholds were examined. With these specifications in hand, the sensor, buzzer, and servo VHDL modules were written. The 100 MHz system clock was converted into 50 MHz (for the sensors and servo) and 25 MHz (for VGA) clocks using a Clocking Wizard IP. A BD (Block Design) was created that included the Clocking Wizard, sensor/buzzer module, and servo module. An ILA (Integrated Logic Analyzer) was added, and the three components were tested for correct operation. Hence, these modules were verified to generate the correct trigger pulses, measure echo durations, produce the appropriate PWM waveform, and emit audible alerts when distances fell below threshold values.

Next, a Comparator module was implemented to evaluate the two sensor outputs and select the closer echo reading for display on the VGA monitor. Then, lookup tables for sine and cosine values were generated externally via Python code that mapped each 0–180° angle step

into Q1.15 number format. The resulting .coe files were loaded into two Block Memory Generator IPs in the block design, with their address inputs tied to the servo's angle output so that real-time sin/cos data was provided for each sweep position.

Finally, the VGA_Radar_Display module was written to accept the distance, sine, and cosine values. These values were converted into pixel coordinates by the module, which then displayed concentric rings and the moving dot on a VGA monitor. Its ports were integrated into the block design and all physical I/O pins were assigned in the XDC file. The complete system was then connected to the Basys 3 board, and scan patterns and buzzer alerts were confirmed to be accurate.

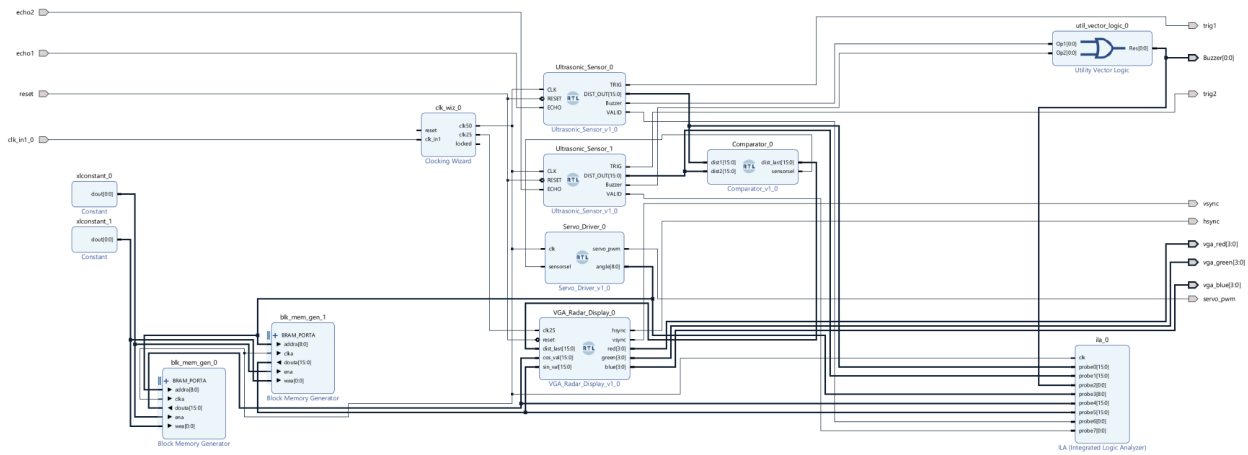


Figure 1: Block design of the entire radar system

Design Specifications

Ultrasonic_Sensor.vhd

The ultrasonic sensor and buzzer module module implements HC-SR04 timing and buzzer alert logic within a single module. Operating on a 50 MHz clock, it issues a 10 µs trigger pulse (*trig*) every 60 ms by transitioning from *IDLE* to *TRIGGER* when a cycle counter reaches *MEASURE_PERIOD*. In the *WAIT_ECHO_HIGH* state it waits for the sonic burst from the

module to take place and the *ECHO* line to assert. In this state, it also has a 20ms timeout possibility in case the sensors are turned off. Then it enters *MEASURE_ECHO*, where it increments an *echo_counter* each clock until *ECHO* signal is back to 0. Upon completion, the raw count is converted to centimeters by dividing by (*CLK_FREQ*/1 000 000) to obtain microseconds, then by *TIME_SCALE* (58 µs/cm), and the result is output on *dist_out* as a 16-bit vector. A one-cycle *VALID* pulse indicates new data availability. When the echo is measured and distance value is calculated, the state goes back to *IDLE*.

Simultaneously, a separate process implements buzzer logic. When the signal *distance_cm* falls below 10 cm, an *alarm_cnt* cycles through its full *ALARM_PERIOD* (*CLK_FREQ*/2), which toggles the *Buzzer* output on for the first half of the period and off for the second half. This results in a constant-frequency beep whenever an object is within the 10 cm threshold. If the object moves beyond that threshold, the buzzer remains silent and the alarm counter resets.

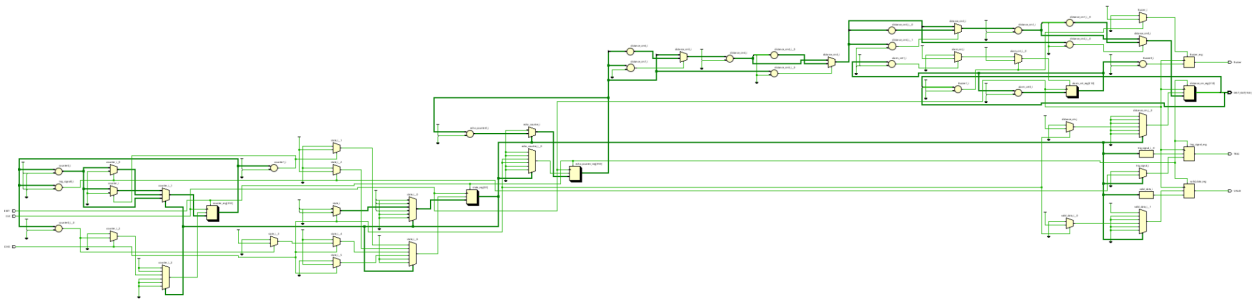


Figure 2: RTL schematic of *Ultrasonic_Sensor.vhd*

Comparator.vhd

The comparator module accepts two 16-bit distance vectors and outputs the smaller value as *dist_last* along with a *sensorsel* output which indicates which sensor detected the closer object. The *dist_last* value is essentially the distance of the object that will be displayed on the

VGA monitor, and *sensorsel* will be used in the servo module to calculate the angle This simple combinational logic ensures that only the nearest echo contributes to the VGA display plot and buzzer logic.

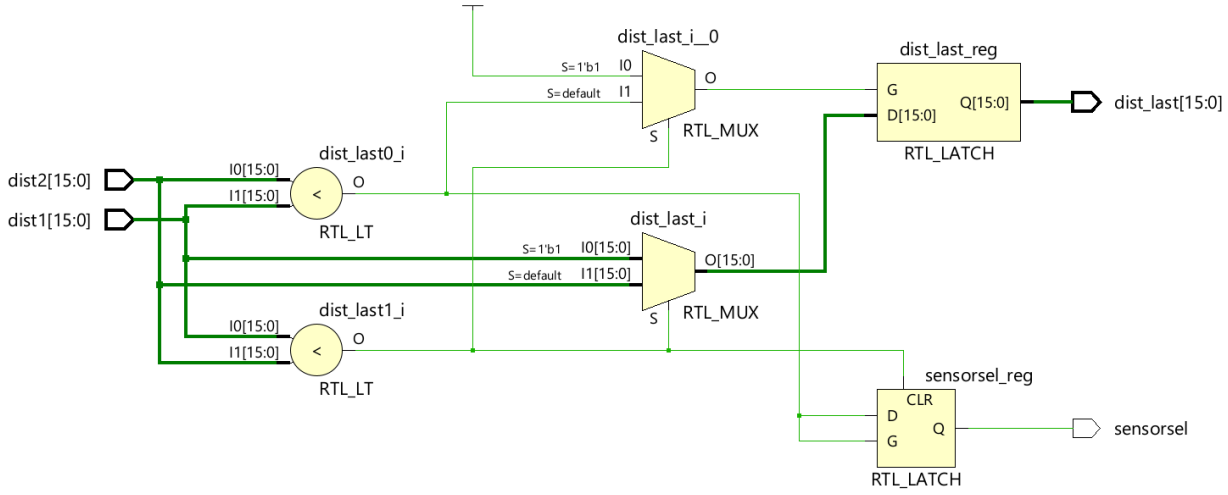


Figure 3: RTL schematic of *Comparator.vhd*

Servo_Driver.vhd

The servo driver module uses the 50 MHz clock to sweep an SG90 mini servo back and forth over 180° by adjusting the pulse width of a 50 Hz PWM signal. A counter runs from 0 to 1 000 000, and while it is below the current *pulse_width*, which ranges from 25 000 counts (0.5 ms) to 125 000 counts (2.5 ms), *servo_pwm* stays high; otherwise it is low. Every 2000 clock cycles, the pulse width increases or decreases by one count, and it reverses direction when it hits the minimum or maximum limits. This sweep count is converted into an angle value (0–180°) and then to a 9-bit unsigned output. The output *sensorsel* from the comparator module is considered while deciding if the calculated angle is acute or obtuse. When *sensorsel* is high, 180 is added to the angle so that the system covers a full 360° sweep for the dual-sensor setup.

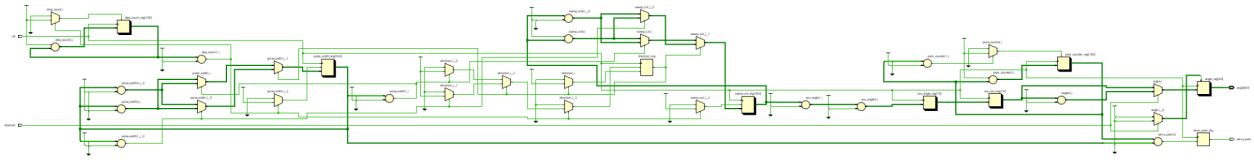


Figure 4: RTL schematic of *Servo_Driver.vhd*

Clocking Wizard IP

The Clocking Wizard IP generates the two required clock domains—50 MHz for sensor/servo/buzzer logic and 25 MHz for VGA timing—by locking onto the board's 100 MHz input.

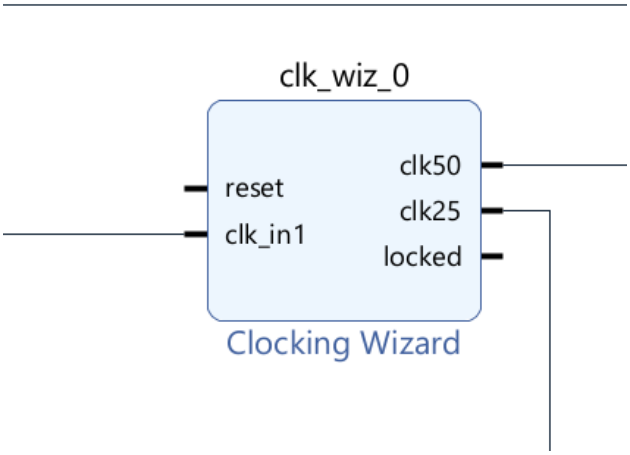


Figure 5: Block design figure of Clocking Wizard IP

Sine and Cosine LUTs

The Lookup Tables were produced via a Python script. It enumerates sine and cosine values for each degree from 0 to 359 in Q1.15 number format. In each loop iteration, the code takes the current angle i (in degrees), converts it to radians with *math.radians(i),* computes the sine or cosine, and then multiplies by $(2^{15} - 1)$ to map the floating-point result (which lies between −1 and +1) into the signed 16-bit Q1.15 number range (−32767…+32767). The values

are truncated into integers and are represented in two's complement form. Finally, it opens two files—sin_lut.coe and cos_lut.coe—and then emits the 360 hex values separated by commas. This .coe file is fed directly into the Block Memory Generator.

Block Memory Generator

The two block memory generator IPs store the precomputed sine and cosine values loaded from the .coe files. Each BRAM is configured with a depth of 360 entries and a data width of 16 bits to match the Q1.15 format. The IP works as a dual-port system: Port A's address is connected to the servo angle signal and Port A's data output drives the cos_val or sin_val nets. Port B remains unused in this case. The BRAMs are initialized at synthesis time by importing the .coe vectors. Hence, in each clock cycle when angle changes, the corresponding fixed-point coefficient is available with one-cycle latency.

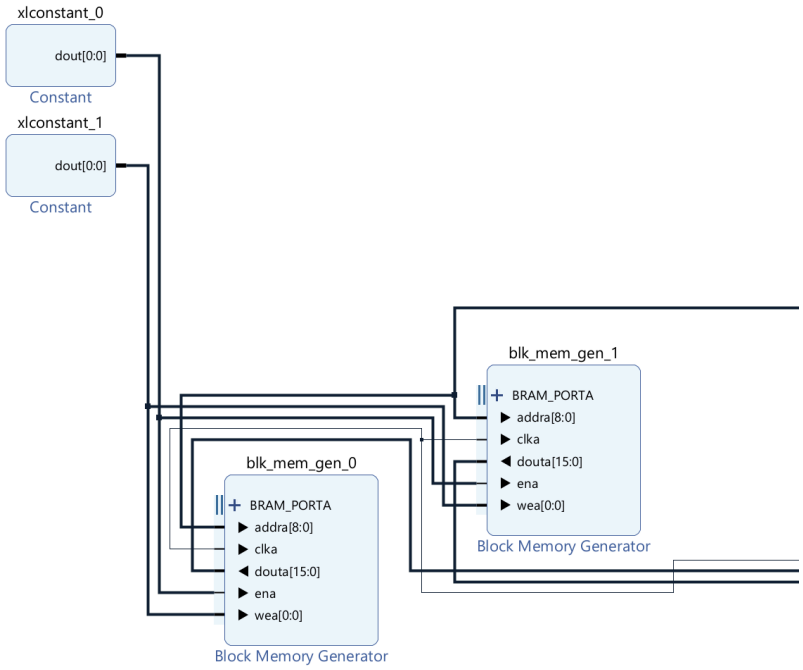


Figure 6: Block design figure of block memory generators for sine and cosine LUTs

VGA_Radar_Display.vhd

The VGA radar display module implements the real-time display of an object on a 640×480 @25 MHz VGA interface. Two nested counters (*h_count* and *v_count*) increment each pixel clock to generate horizontal and vertical positions with visible assertions when counters lie within the active display region. For each pixel, the module computes offsets $dx = h_{count} - CENTER_X$ and $dy = v_{count} - CENTER_Y$, then forms $dist_{sq} = dx \times dx + dy \times dy$. It tests these *dist_sq* values against each entry of the *RADII* array (60, 120, 180, 240) with a tolerance of ±radius to draw the concentric green rings. Simultaneously, the module reads *dist_last*, *cos_val*, and *sin_val* values and converts distance to *radius_pix* and computes $dot_x = CENTER_X + (cos_{val} \times radius_{pix})/2^{15}$ and $dot_y = CENTER_Y - (sin_{val} \times radius_{pix})/2^{15}$. It then asserts *dot_on* when the current pixel lies within *DOT_RADIUS* of (*dot_x*, *dot_y*), coloring it red. Color outputs (red, green, blue) are multiplexed such that the moving dot overrides ring pixels, which in turn override background. Meanwhile, *hsync* and *vsync* signals are generated by comparing counters against front porch and sync pulse constants, which ensures that the VGA's timing is accurate. The entire process introduces only one clock cycle of latency between data inputs and pixel output.

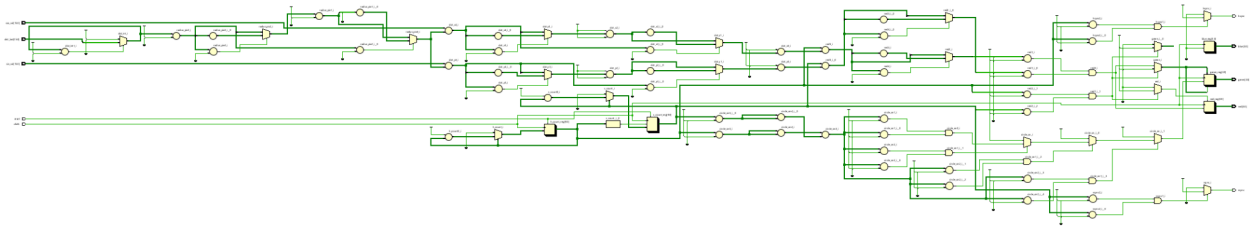


Figure 7: RTL schematic of *VGA_Radar_Display.vhd*

Integrated Logic Analyzer (ILA)

The ILA helps the user observe the numerical values of any input or output they want to see, including *dist_last*, *angle*, *cos_val*, *sin_val*, *trigger* and more during runtime. This IP was very important during early sensor testing when no VGA output was available to visualize distances. Using this IP I could verify that the ultrasonic echo measurements, pulse widths, and distance conversions behaved as expected.

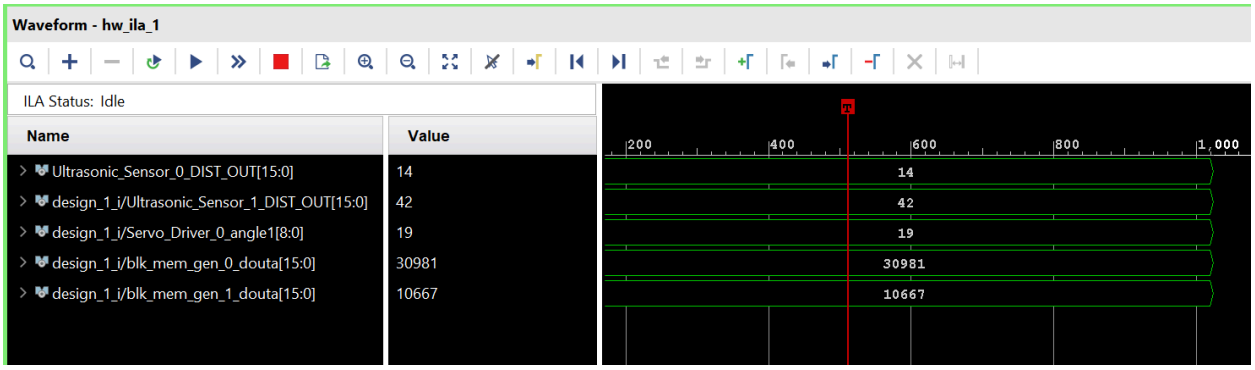


Figure 8: An example ILA output table

Results

On the Basys 3 board, the radar executed smooth 360° sweeps and updated the VGA display in real time with concentric rings and a moving dot. The buzzer responded instantly whenever an object crossed the threshold—all with no noticeable lag. Some example results can be observed in Figures 9-17.

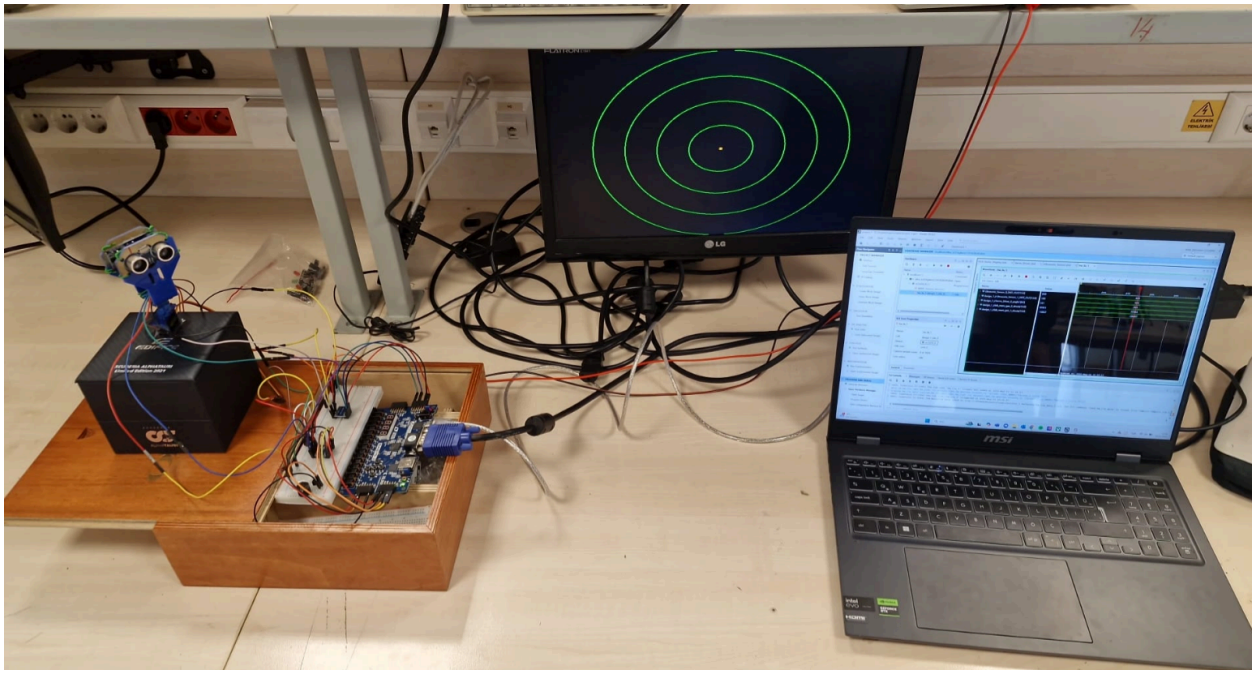
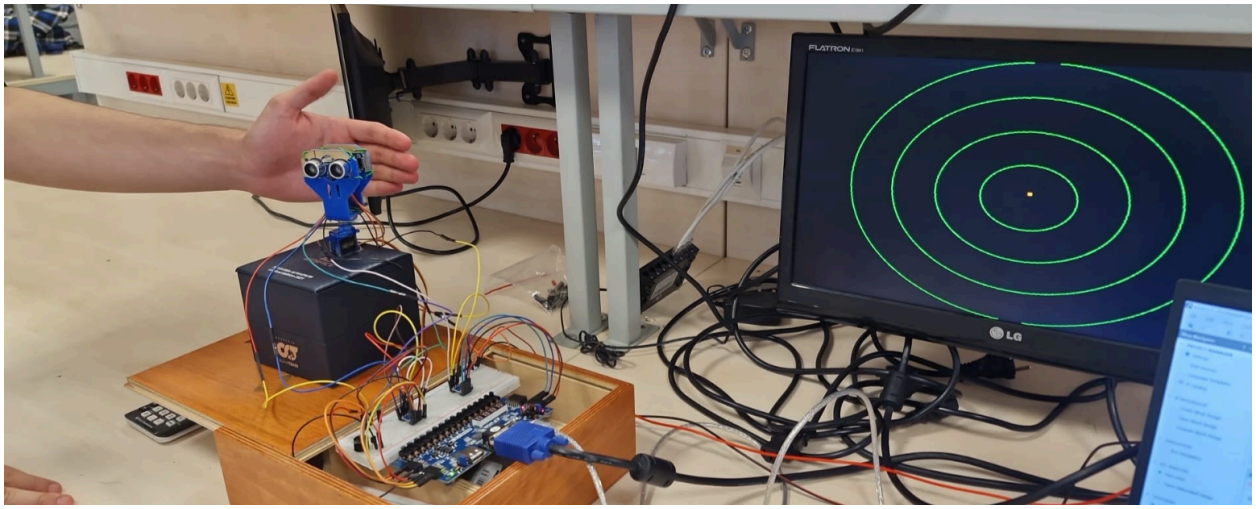
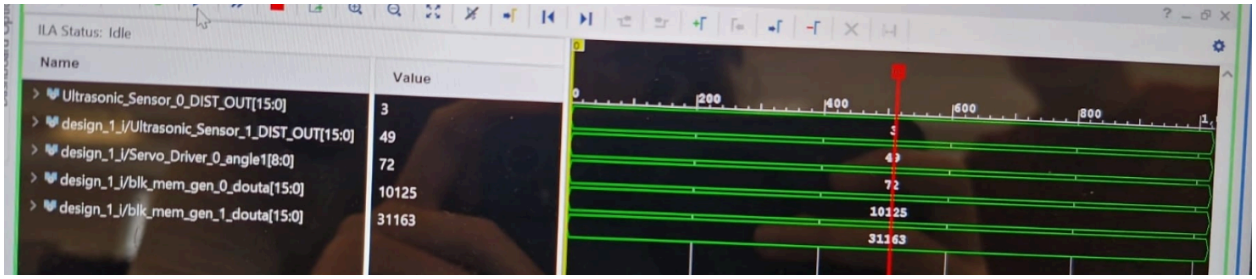
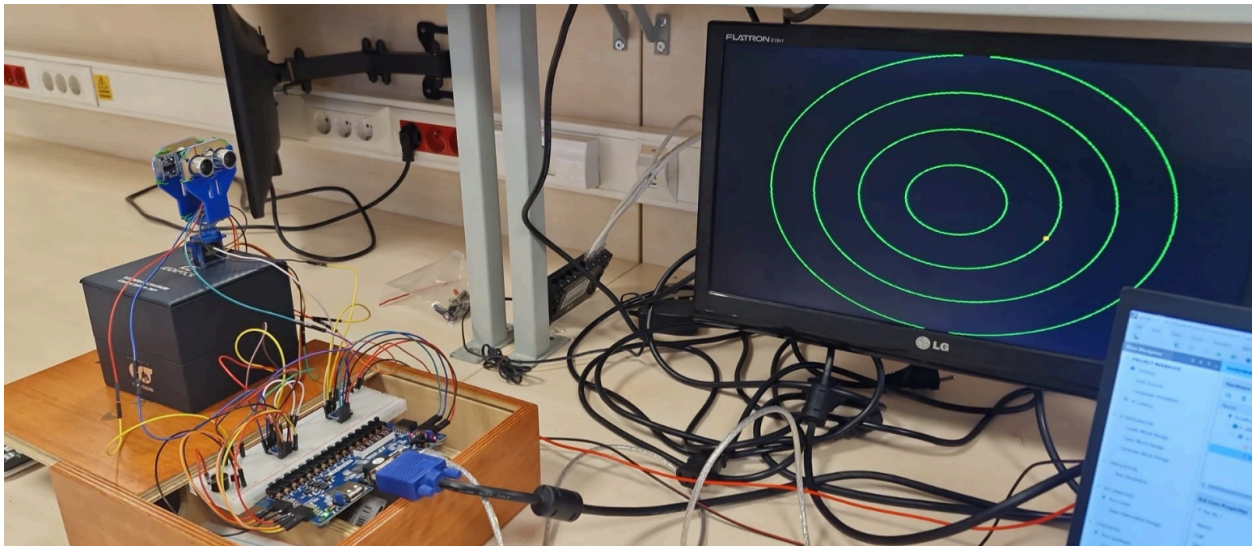


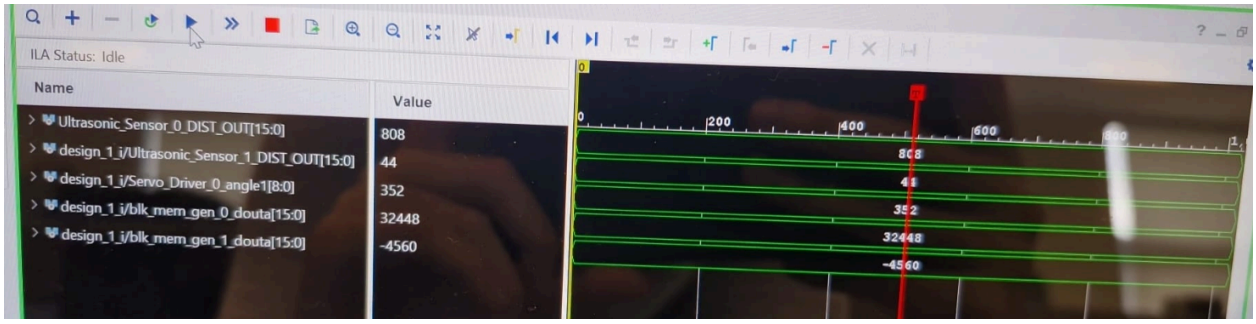
Figure 10: Initially turned on system



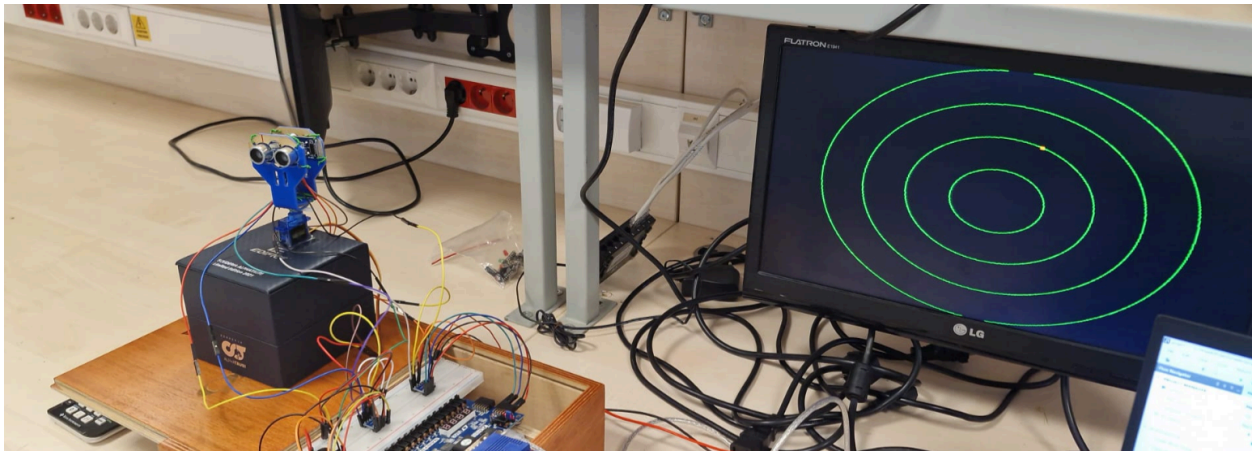


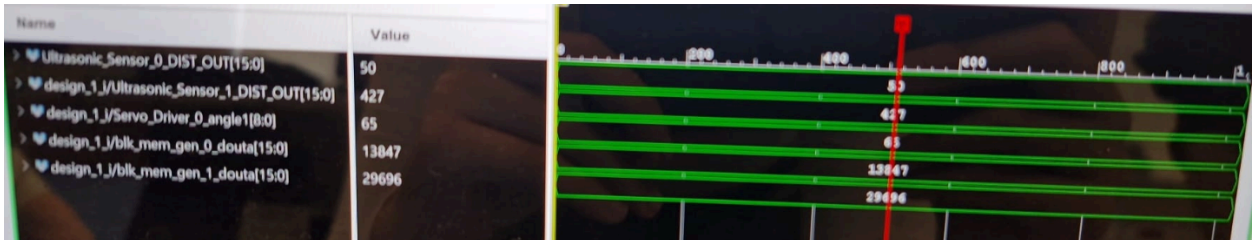
Figures 11-12: Closest object at 72° angle. Buzzer = 1, dist_last = 3



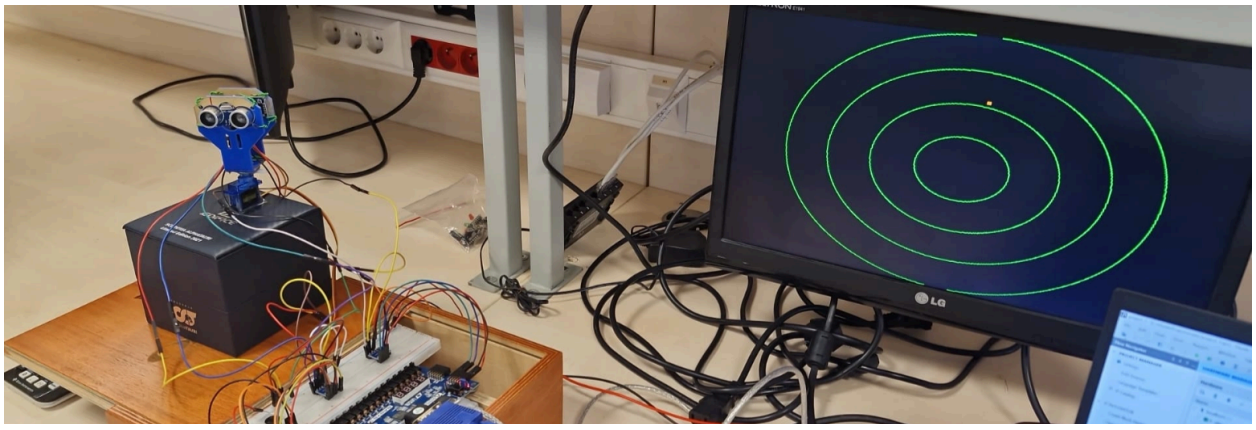


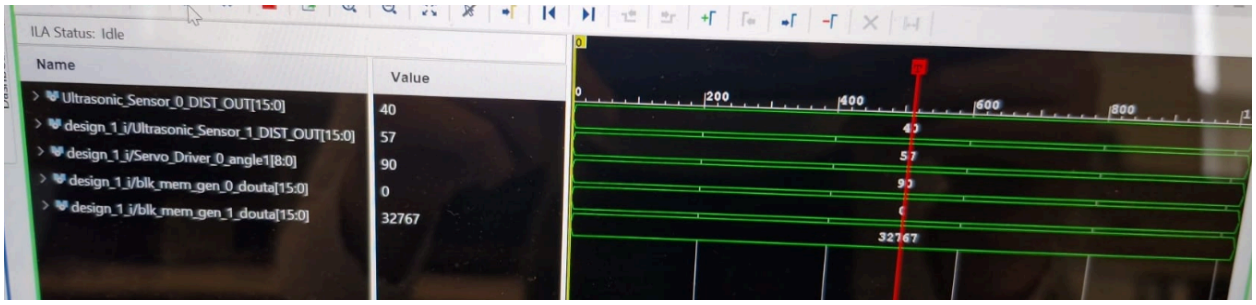
Figures 12-13: Closest object at 352° angle. Buzzer = 0, dist_last = 44





Figures 14-15: Closest object at 65° angle. Buzzer = 0, dist_last = 50





Figures 16-17: Closest object at 90° angle. Buzzer = 0, dist_last = 40

Conclusion

In this project, I built a fully digital ultrasonic radar on the Basys3 board by writing VHDL modules for the HC-SR04 sensors, buzzer, servo, comparator and VGA display, and tying them together with Xilinx IP modules and block designs. On a 50 MHz clock, the sensor module issues 10 µs triggers every 60 ms, measures echo pulses in microseconds and converts them to centimeters; a second process drives the buzzer when objects get closer than 10 cm. A simple comparator picks the smaller of two distance readings and feeds a 9-bit angle selector into both

the servo driver and the VGA display. The servo driver generates a 50 Hz PWM whose cycle is between 0.5 ms and 2.5 ms, and sweeps the sensors through 360° and reports its position as an angle. Externally generated Q1.15 sine and cosine lookup tables are stored in BRAMs and, together with the measured distance, are converted into (x,y) pixels to draw concentric range rings and a moving red dot on a 640×480 VGA screen. Throughout the process I used an ILA to verify trigger timing, echo counting and PWM behavior before VGA output was available.

The toughest part of the project was mapping the servo's physical angle into precise fixed-point sine and cosine values. Writing the Python script to generate 360 Q1.15 entries, checking two's-complement accuracy, and then aligning those values in real time so the radar dot appeared at exactly the right screen position was the most challenging part of my work, but once the LUTs and BRAMs were in place, the display matched the real-world scan perfectly.

Overall, I ran into a few errors during testing. First, the HC-SR04 echo pulses would sometimes jitter a microsecond or two which is about 1–2 cm of error, so my distance readings would vary occasionally. To fix it, I'd throw a simple moving-average filter on the last few echoes and double-check the 58 μs/cm scale against a ruler. Second, the servo didn't move in perfect steps since equal PWM increases didn't always translate to equal angles, so the movement of the object on the monitor is sometimes sharper than intended. In future projects I could either remap the PWM-to-angle LUT based on a quick servo calibration or add an encoder for feedback. Finally, my Q1.15 sine-cosine tables caused tiny rounding errors on the VGA plot, which caused slight errors on the display. Increasing the LUT values to more entries such as 720 or 1080 entries will help fix this error in future projects.

References

Digilent Inc. (n.d.). *Basys 3™ FPGA Board Reference Manual* (Rev. C). Retrieved May 14, 2025, from https://digilent.com/reference/_media/basys3%3Abasys3_rm.pdf

Friendlywire. (n.d.). *SG90 Micro Servo Datasheet*. Retrieved May 14, 2025, from https://www.friendlywire.com/projects/ne555-servo-safe/SG90-datasheet.pdf

Python Software Foundation. (2024). *math — Mathematical functions* (Python 3.13.3 documentation). Retrieved May 14, 2025, from https://docs.python.org/3/library/math.html

SparkFun Electronics. (n.d.). *HC-SR04 Ultrasonic Sensor Module Datasheet*. Retrieved May 14, 2025, from https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

Xilinx, Inc. (2022). *Block Memory Generator LogiCORE™ IP Product Guide (PG058, v8.4)*. Retrieved May 14, 2025, from https://docs.amd.com/v/u/en-US/pg058-blk-mem-gen

Xilinx, Inc. (2022). *Clocking Wizard LogiCORE™ IP Product Guide (PG065, v6.0)*. Retrieved May 14, 2025, from https://docs.amd.com/r/en-US/pg065-clk-wiz

Xilinx, Inc. (2022). *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (UG994, v2022.1)*. Retrieved May 14, 2025, from https://docs.amd.com/r/en-US/ug994-vivado-ip-subsystems

Appendix

Ultasonic_Sensor.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity Ultrasonic_Sensor is
    Port (
        CLK         : in  std_logic; -- 50 MHz
        RESET       : in  std_logic;
        TRIG        : out std_logic;
        ECHO        : in  std_logic;
        DIST_OUT    : out std_logic_vector(15 downto 0);
        Buzzer      : out std_logic;
        VALID       : out std_logic
    );
end Ultrasonic_Sensor;

architecture Behavioral of Ultrasonic_Sensor is

    constant CLK_FREQ        : integer := 50000000;
    constant TRIG_PULSE_CYC  : integer := 500;        -- 10us @ 50MHz
    constant MEASURE_PERIOD  : integer := 3000000;     -- 60ms @
50MHz
    constant TIME_SCALE      : integer := 58;          -- uS/58 = cm
    constant ALARM_PERIOD    : integer := CLK_FREQ/2;

    type state_type is (IDLE, TRIGGER, WAIT_ECHO_HIGH, MEASURE_ECHO,
DONE);
    signal state : state_type := IDLE;

    signal counter           : integer := 0;
    signal trig_signal       : std_logic := '0';
    signal echo_counter      : integer := 0;
    signal distance_cm       : integer := 0;
    signal valid_data        : std_logic := '0';
```

```
    signal alarm_cnt         : integer := 0;

begin

    process(CLK, RESET)
    begin
        if RESET = '1' then
            state        <= IDLE;
            counter      <= 0;
            echo_counter <= 0;
            distance_cm  <= 0;
            trig_signal  <= '0';
            valid_data   <= '0';
        elsif rising_edge(CLK) then
            case state is

                when IDLE =>
                    trig_signal  <= '0';
                    valid_data   <= '0';
                    if counter < MEASURE_PERIOD then
                        counter <= counter + 1;
                    else
                        counter <= 0;
                        state <= TRIGGER;
                    end if;

                when TRIGGER =>
                    if counter < TRIG_PULSE_CYC then
                        trig_signal <= '1';
                        counter <= counter + 1;
                    else
                        trig_signal <= '0';
                        counter <= 0;
                        state <= WAIT_ECHO_HIGH;
                    end if;

                when WAIT_ECHO_HIGH =>
                    if ECHO = '1' then
                        echo_counter <= 0;
```

```
                    state <= MEASURE_ECHO;
                elsif counter < 1000000 then -- 20ms timeout
                    counter <= counter + 1;
                else
                    state <= DONE;
                end if;

            when MEASURE_ECHO =>
                if ECHO = '1' then
                    echo_counter <= echo_counter + 1;
                else
                    distance_cm <= echo_counter / (CLK_FREQ /
1000000) / TIME_SCALE;
                    valid_data  <= '1';
                    state <= DONE;
                end if;

            when DONE =>
                state <= IDLE;

            when others =>
                state <= IDLE;

        end case;
    end if;
end process;

  process(CLK)
  begin
    if rising_edge(CLK) then
      if distance_cm < 10 then
        if alarm_cnt < ALARM_PERIOD/2 then
          Buzzer <= '1';
        else
          Buzzer <= '0';
        end if;

        if alarm_cnt < ALARM_PERIOD-1 then
          alarm_cnt <= alarm_cnt + 1;
```

```
            else
              alarm_cnt <= 0;
            end if;
          else
            alarm_cnt <= 0;
            Buzzer    <= '0';
          end if;
        end if;
      end process;

    TRIG     <= trig_signal;
    DIST_OUT <= conv_std_logic_vector(distance_cm, 16);
    VALID    <= valid_data;

end Behavioral;
```

Servo_Driver.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Servo_Driver is
    Port (
        clk       : in  std_logic;   -- 50 MHz system clock
        sensorsel : in std_logic;
        servo_pwm : out std_logic;    -- PWM output to servo
        angle : out unsigned(8 downto 0)
    );
end Servo_Driver;

architecture Behavioral of Servo_Driver is
    constant CLOCK_FREQ   : integer := 50000000;          -- 50 MHz
    constant PWM_FREQ     : integer := 50;                -- 50 Hz
(20 ms period)
    constant PWM_PERIOD   : integer := CLOCK_FREQ / PWM_FREQ;  --
1 000 000 counts
```

```
    constant MIN_PULSE    : integer := (PWM_PERIOD / 20000) * 500;
-- 0.5 ms  -> 25 000
    constant MAX_PULSE    : integer := (PWM_PERIOD / 20000) * 2500;
-- 2.5 ms -> 125 000

    constant STEP_SIZE    : integer := 1;
    constant STEP_DIVIDER : integer := 2000;

    constant SWEEP_MAX : integer := (MAX_PULSE - MIN_PULSE) /
STEP_SIZE;

    signal pwm_counter  : integer range 0 to PWM_PERIOD := 0;
    signal pulse_width  : integer range MIN_PULSE to MAX_PULSE :=
MIN_PULSE;
    signal direction    : std_logic := '1';    -- '1'=increasing,
'0'=decreasing
    signal step_count   : integer range 0 to STEP_DIVIDER := 0;
    signal sweep_cnt : integer range 0 to SWEEP_MAX := 0;
    signal raw_angle : integer range 0 to 180;
    signal raw_vec   : unsigned(8 downto 0);

begin
    process(clk)
    begin
        if rising_edge(clk) then
            if pwm_counter < PWM_PERIOD then
                pwm_counter <= pwm_counter + 1;
            else
                pwm_counter <= 0;
            end if;

            if pwm_counter < pulse_width then
                servo_pwm <= '1';
            else
                servo_pwm <= '0';
            end if;

            if step_count < STEP_DIVIDER-1 then
                step_count <= step_count + 1;
```

```
            else
                step_count <= 0;
                if direction = '1' then
                    if pulse_width < MAX_PULSE then
                        pulse_width <= pulse_width + STEP_SIZE;
                        sweep_cnt   <= sweep_cnt + 1;
                    else
                        direction   <= '0';
                        pulse_width <= pulse_width - STEP_SIZE;
                        sweep_cnt   <= sweep_cnt - 1;
                    end if;
                 else
                    if pulse_width > MIN_PULSE then
                        pulse_width <= pulse_width - STEP_SIZE;
                        sweep_cnt   <= sweep_cnt - 1;
                    else
                        direction   <= '1';
                        pulse_width <= pulse_width + STEP_SIZE;
                        sweep_cnt   <= sweep_cnt + 1;
                    end if;
                 end if;
             end if;


        raw_angle <= (sweep_cnt * 180) / SWEEP_MAX;           --
0...180
        raw_vec   <= to_unsigned(raw_angle, raw_vec'length); -- 9
bits

        if sensorsel = '0' then
          angle <= raw_vec;                                  -- 0...180°
        else
          angle <= raw_vec + to_unsigned(180, 9);        --
180...360°
        end if;

        end if;
    end process;
end Behavioral;
```

Comparator.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_unsigned;

entity Comparator is
    Port ( dist1 : in std_logic_vector(15 downto 0);
           dist2 : in std_logic_vector(15 downto 0);
           dist_last : out std_logic_vector(15 downto 0);
           sensorsel : out std_logic
    );
end Comparator;

architecture Behavioral of comparator is

begin

process(dist1, dist2)
begin
    if dist1 < dist2 then
        dist_last <= dist1;
        sensorsel <= '0';

    elsif dist2 < dist1 then
        dist_last <= dist2;
        sensorsel <= '1';
    end if;
end process;
end Behavioral;
```

VGA_Radar_Display.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity VGA_Radar_Display is
```

```
    Port (
        clk25          : in  std_logic;    -- 25 MHz pixel clock
        reset          : in  std_logic;
        dist_last      : in  std_logic_vector(15 downto 0);
        cos_val        : in  std_logic_vector(15 downto 0);  -- Q1.15
        sin_val        : in  std_logic_vector(15 downto 0);  -- Q1.15
        hsync          : out std_logic;
        vsync          : out std_logic;
        red            : out std_logic_vector(3 downto 0);
        green          : out std_logic_vector(3 downto 0);
        blue           : out std_logic_vector(3 downto 0)
    );
end VGA_Radar_Display;

architecture Behavioral of VGA_Radar_Display is
    constant H_VISIBLE       : integer := 640;
    constant H_FRONT_PORCH   : integer := 16;
    constant H_SYNC_PULSE    : integer := 96;
    constant H_BACK_PORCH    : integer := 48;
    constant H_TOTAL         : integer := H_VISIBLE + H_FRONT_PORCH +
H_SYNC_PULSE + H_BACK_PORCH;

    constant V_VISIBLE       : integer := 480;
    constant V_FRONT_PORCH   : integer := 10;
    constant V_SYNC_PULSE    : integer := 2;
    constant V_BACK_PORCH    : integer := 33;
    constant V_TOTAL         : integer := V_VISIBLE + V_FRONT_PORCH +
V_SYNC_PULSE + V_BACK_PORCH;

    constant CENTER_X        : integer := H_VISIBLE/2;
    constant CENTER_Y        : integer := V_VISIBLE/2;

    type radii_array is array (0 to 3) of integer;
    constant RADII           : radii_array := (60, 120, 180, 240);

    constant DISPLAY_RANGE_CM: integer := 100;

    constant DOT_RADIUS      : integer := 2;
```

```
    signal h_count    : integer range 0 to H_TOTAL-1 := 0;
    signal v_count    : integer range 0 to V_TOTAL-1 := 0;
    signal visible    : std_logic;
    signal dist_int   : integer;
    signal radius_pix : integer;
    signal cos_s, sin_s : signed(15 downto 0);
    signal dot_x, dot_y : integer;
begin
  process(clk25, reset)
  begin
    if reset = '1' then
      h_count <= 0; v_count <= 0;
    elsif rising_edge(clk25) then
      if h_count = H_TOTAL-1 then
        h_count <= 0;
        if v_count = V_TOTAL-1 then v_count <= 0;
        else                        v_count <= v_count + 1;
        end if;
      else
        h_count <= h_count + 1;
      end if;
    end if;
  end process;

  hsync   <= '0' when (h_count >= H_VISIBLE+H_FRONT_PORCH
                   and h_count <
H_VISIBLE+H_FRONT_PORCH+H_SYNC_PULSE)
            else '1';
  vsync   <= '0' when (v_count >= V_VISIBLE+V_FRONT_PORCH
                   and v_count <
V_VISIBLE+V_FRONT_PORCH+V_SYNC_PULSE)
            else '1';
  visible <= '1' when (h_count < H_VISIBLE and v_count < V_VISIBLE)
else '0';

  process(dist_last, cos_val, sin_val)
  begin
    dist_int <= to_integer(unsigned(dist_last));
    if dist_int > DISPLAY_RANGE_CM then
```

```
      dist_int <= DISPLAY_RANGE_CM;
    end if;
    radius_pix <= dist_int * RADII(RADII'length-1) /
DISPLAY_RANGE_CM;

    cos_s <= signed(cos_val);
    sin_s <= signed(sin_val);

    dot_x <= CENTER_X + (to_integer(cos_s) * radius_pix) / 2**15;
    dot_y <= CENTER_Y - (to_integer(sin_s) * radius_pix) / 2**15;
  end process;


  process(clk25)
    variable dx, dy    : integer;
    variable dist_sq   : integer;
    variable circle_on : boolean;
    variable dot_on    : boolean;
  begin
    if rising_edge(clk25) then
      if visible = '1' then
        dx := h_count - CENTER_X;
        dy := v_count - CENTER_Y;
        dist_sq := dx*dx + dy*dy;

        circle_on := false;
        for i in RADII'range loop
          if dist_sq >= RADII(i)*RADII(i) - RADII(i)
          and dist_sq <= RADII(i)*RADII(i) + RADII(i) then
            circle_on := true;
          end if;
        end loop;

        dot_on := (abs(h_count - dot_x) <= DOT_RADIUS
                   and abs(v_count - dot_y) <= DOT_RADIUS);

        if dot_on then
          red   <= "1111"; green <= "1000"; blue  <= "0000";
        elsif circle_on then
```

```
          red   <= "0000"; green <= "1111"; blue  <= "0000";
        else
          red   <= "0000"; green <= "0000"; blue  <= "0000";
        end if;
      else
        red   <= "0000"; green <= "0000"; blue  <= "0000";
      end if;
    end if;
  end process;
end Behavioral;
```

lut.py

```
import math

N = 360
with open('sin_lut.coe','w') as f:
    f.write('memory_initialization_radix=10;\n')
    f.write('memory_initialization_vector=\n')
    for i in range(N):
        v = int((2**15 - 1) * math.sin(math.radians(i)))
        f.write(f'{v}' + (',' if i < N-1 else ';\n'))

with open('cos_lut.coe','w') as f:
    f.write('memory_initialization_radix=10;\n')
    f.write('memory_initialization_vector=\n')
    for i in range(N):
        v = int((2**15 - 1) * math.cos(math.radians(i)))
        f.write(f'{v}' + (',' if i < N-1 else ';\n'))
```

design_1_wrapper.v

```
`timescale 1 ps / 1 ps

module design_1_wrapper
   (Buzzer,
    clk_in1_0,
```

```
    echo1,
    echo2,
    hsync,
    reset,
    servo_pwm,
    trig1,
    trig2,
    vga_blue,
    vga_green,
    vga_red,
    vsync);
  output [0:0]Buzzer;
  input clk_in1_0;
  input echo1;
  input echo2;
  output hsync;
  input reset;
  output servo_pwm;
  output trig1;
  output trig2;
  output [3:0]vga_blue;
  output [3:0]vga_green;
  output [3:0]vga_red;
  output vsync;

  wire [0:0]Buzzer;
  wire clk_in1_0;
  wire echo1;
  wire echo2;
  wire hsync;
  wire reset;
  wire servo_pwm;
  wire trig1;
  wire trig2;
  wire [3:0]vga_blue;
  wire [3:0]vga_green;
  wire [3:0]vga_red;
  wire vsync;
```

```
  design_1 design_1_i
       (.Buzzer(Buzzer),
        .clk_in1_0(clk_in1_0),
        .echo1(echo1),
        .echo2(echo2),
        .hsync(hsync),
        .reset(reset),
        .servo_pwm(servo_pwm),
        .trig1(trig1),
        .trig2(trig2),
        .vga_blue(vga_blue),
        .vga_green(vga_green),
        .vga_red(vga_red),
        .vsync(vsync));
endmodule
```