Ultrasonic-Based Flight Obstacle Avoidance Game

Xu H. Kong
Electrical and Electronic Engineering department
Southern University of Science and Technology
Shenzhen, China
12012112@mail.sustech.edu.cn

Yu Wang
Electrical and Electronic Engineering department
Southern University of Science and Technology
Shenzhen, China
12112725@mail.sustech.edu.cn

I. Introduction–worked together

In this project, we utilized the Nexys4 DDR development board, combined with VHDL language programming and two PMOD peripherals: a VGA display and an HC-SR04 ultrasonic distance sensor, to successfully develop an innovative flight avoidance game inspired by the popular mobile game Flappy Bird. Through the integration of hardware and software, we designed a game system where the altitude control of the aircraft is achieved by real-time detection of the relative distance between the player's hand and the HC-SR04 ultrasonic sensor.

This project not only demonstrates complex digital logic design capabilities but also embodies the interactivity of embedded system design. Players control the aircraft on the screen by physically moving their hand, ascending or descending to avoid obstacles in the path. This intuitive mode of interaction greatly enhances the user experience, making the game not only a visual delight but also a form of physical participation. Through the VGA protocol, the graphical information of the aircraft and obstacles is clearly displayed on the screen, providing players with rich visual feedback.

II. Global inputs and outputs–worked together

The global inputs and outputs are as follows:

Input signals:

- clk: system clock;
- reset: asynchronous reset signal for system initialization;
- playAgain: the button to play the game again;
- freeze: the switch to pause the game;
- echo: receive signal from ultrasonic module

Output signals:

- hsync: an output signal used in video displays to synchronize the horizontal scanning of the display. It indicates the end of one row of pixels and the start of the next.
- vsync: an output signal used in video displays to synchronize the vertical scanning of the display. It indicates the end of one frame and the start of the next.
- red,green,blue: a vector representing the red/green/blue color component of the pixel being displayed. 4 bits.
- trig: send trigger signal to ultrasonic module

III. Project Structure–worked by Kong

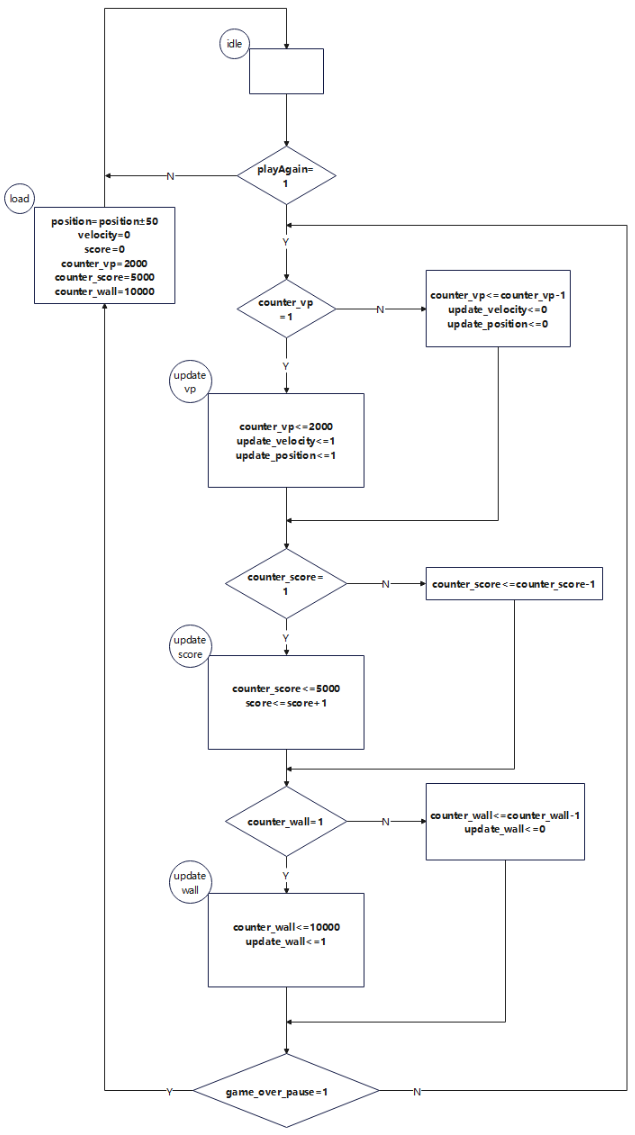


Fig. 1. ASM chart of game logic

The ASM (Algorithmic State Machine) chart illustrates the control flow of a game. Here is a simplified analysis:

- The system starts in an *idle* state.
- The main loop checks if the *play again* register is set to be 1. If yes, it proceeds; otherwise, it waits.
- It updates velocity and position every 2000 clocks.
- It increments the score every 5000 clocks
- It updates the wall status every 10000 clocks.
- If the *game_over_pause* register is 1, the game ends, then it load the initial values and goes into *idle* state.

The system continuously cycles through these steps until the game ends or restarts.

The following is the corresponding logic control code:

```
process ( video_on )
        variable counter : integer := 0;
        variable vel_counter : integer := 0;
        variable wall_counter: integer := 0;
        variable score_counter: integer := 0;
        variable game_over_counter: integer :=
             0;
    begin
        if game_over_pause = '1' then
            score <= 0;
        elsif (rising_edge(video_on) and
            freeze = '0') and game_over_pause
            = '0' then
            if start_pause = '0' then
                counter := counter + 1;
                vel_counter := vel_counter +
                    1;
                wall_counter := wall_counter +
                     1;
                score_counter := score_counter
                     + 1;

                if counter > 2000 then --
                    update postion every 2000
                    clocks
                    counter := 0;
                    update_pos <= '1';
                else
                    update_pos <= '0';
                end if;
                if vel_counter > 2000 then --
                    update velocity every 2000
                     clocks
                    vel_counter := 0;
                    update_vel <= '1';
                else
                    update_vel <= '0';
                end if;
                if wall_counter > 10000 then
                    --walls increment every
                    10000 clocks
                    wall_counter := 0;
                    update_walls <= '1';
                else
                    update_walls <= '0';
                end if;
                if score_counter > 5000 then
                    --scores incrment every
                    5000 clocks
                    score <= score + 1;
                    score1 <= score mod 10;
                    score2 <= (score / 10) mod
                         10;
                    score3 <= (score / 100)
                        mod 10;
                    score4 <= (score / 1000)
                        mod 10;
                    score_counter := 0;
                end if;
            elsif btn = '1' then
                start_pause <= '0';
            end if;
        end if;
    end process;
```

The following is the structure of this program:

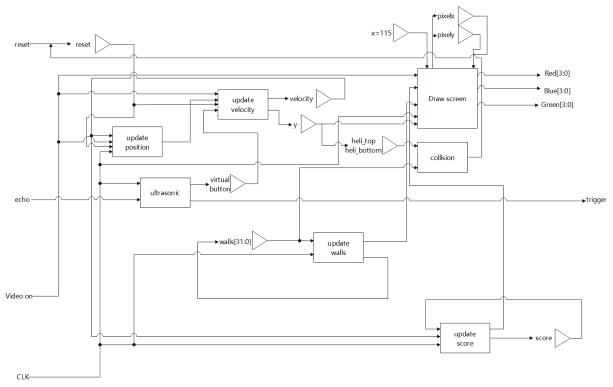


Fig. 2. game structure

IV. Project Logic

A. update position—-worked by Kong

This part has three functions:

1. Game Reset Detection:

- If the game ends (`game_over_pause = '1'`) and the player presses the restart button (`playAgain = '1'`), then reset the game state, setting `game_over_pause` to '0'.
- If the game ends but the player does not press the restart button, do nothing.

2. Update Helicopter Position:

- If the `update_pos` signal is triggered on the rising edge (`rising_edge(update_pos)`), then update the vertical position y of the helicopter by adding the current vertical speed `velocity_y`.

3. Collision Detection:

- Check if the helicopter's bottom position (`heli_bottom`) exceeds the top of the current cave plus the height of the walls (`cave_width + walls(6)`). If so, the helicopter collides with the top of the cave, set the helicopter's position to `walls(7) + 50`, and set the game state to end (`game_over_pause <= '1'`).
- Check if the helicopter's top position (`heli_top`) is lower than the top position of the current wall (`walls(6)`). If so, the helicopter collides with the

bottom of the cave, set the helicopter's position to walls(7) + 50, and set the game state to end (game_over_pause <= '1').

The following is the corresponding code:

```
    process (playAgain, update_pos, video_on)
    begin
        if game_over_pause = '1' and playAgain
             = '1' then
            game_over_pause <= '0';
        elsif game_over_pause = '1' and
            playAgain = '0' then
        elsif rising_edge(update_pos) then
            y <= y + velocity_y;
            if (heli_bottom >= cave_width +
                walls(6)) then   -- calculate
                collision with walls
                y <= walls(7) + 50;
                game_over_pause <= '1';
            elsif (heli_top <= walls(6))then
                y <= walls(7) + 50;
                game_over_pause <= '1';
            end if;
        end if;
    end process;
```

B. *update velocity—-worked by Kong*

- **Maximum Upward Speed:** TVU
- **Maximum Downward Speed:** TVD(both are positive values)
- **When the virtual button is pressed:**
  - If the current vertical speed is greater than the terminal upward speed -TVU, decrease the vertical speed (i.e., increase the upward speed, as upward speed is negative), and the helicopter's ascent speed increases.
- **When the virtual button is not pressed:**
  - If the current vertical speed is less than the terminal downward speed TVD, increase the vertical speed (i.e., increase the descent speed), and the helicopter's descent speed increases.

The following is the corresponding code:

```
    process (update_vel)
    begin
        if rising_edge(update_pos) then
            if btn = '1' then
                if velocity_y > -TVU then
                    velocity_y <= velocity_y -
                         1;
                end if;
            else
                if velocity_y < TVD then
                    velocity_y <= velocity_y +
                         1;
                end if;
            end if;
        end if;
    end process;
```

C. *update walls—-worked by Wang*

The wall consists of two parts: wall position and cave width.

- When the update_walls signal is triggered on the rising edge, the wall will move to the left constantly, and the position of the rightmost wall will be adjusted based on the pseudorandom number and cave width.
- When the cave width is too narrow, assign it a larger value and increase the wall width by 1 unit with each update; when the cave width is too wide, assign it a smaller value and decrease the wall width by 1 unit with each update.
- When the wall position is too low, assign it a larger value and use a pseudorandom number to increase it within the range (-10, 30); when the position is too high, assign it a smaller value and use a pseudorandom number to decrease it within the range (-30, 10).

The following is the corresponding code:

```
process (update_walls)
    begin
        if rising_edge(update_walls)  then
            if (cave_width < 250) then
                cave_width <= 255;
                general_width_up <= '1';
            elsif (cave_width > 370) then
                cave_width <= 365;
                general_width_up <= '0';
            elsif (general_width_up = '1')
                then
                cave_width <= cave_width + 1;
            else
                cave_width <= cave_width - 1;
            end if;
            for i in 1 to 31 loop
                walls(i - 1) <= walls(i);
            end loop;
            if(walls(31) < 31) then
                general_up <= '1';
                walls(31) <= 35;
            elsif (walls(31) >= 230) then
                general_up <= '0';
                 walls(31) <= 225;
            elsif(general_up = '1')then
                 walls(31) <= walls(31)+ ((
                     walls(2) * walls(19) +
                     walls(25) * 13) mod 40)
                     -10; --add value between
                     -10 and 30
                 if((heli_top + walls(2))*13
                     mod 10 = 1) then--10 % of
                      the time change general
                     wall direction
                    general_up <= '0';
                    end if;
            else
                walls(31) <= walls(31)- ((
                    walls(2) * walls(19) +
                    walls(25) * 13) mod 40)
                    +10; --add value between
                    -30 and 10
                if((heli_top + walls(2))*13
                    mod 10 = 1) then --10 % of
                     the time change general
                    wall direction
                    general_up <= '1';
                    end if;
           end if;
        end if;
    end process;
```

D. update and generate blocks—-worked by Wang

The obstacle consists of two parts: generate_block and update_block.

- The logic for updating the obstacle is similar to updating the wall. When the update_block signal is triggered on the rising edge, the obstacle will move to the left and be adjusted based on the wall position.
- The generate_block is used to determine whether an obstacle needs to be generated. At certain clock cycles, generate_block is set to 1 and maintained for two clock cycles to ensure the width of the obstacle and to avoid the obstacle appearing too frequently.

The following is the corresponding code:

```
process(generate_block, update_blocks)
    begin
        if rising_edge(update_blocks) then
            for i in 1 to 31 loop
                blocks(i-1) <= blocks(i);
            end loop;
        end if;
        if generate_block = '1' then
            if blocks(31) = 0 then
                blocks(31) <= 150;
            end if;
            if walls(31) <= 230 then
                blocks(31) <= blocks(31)- ((
                    walls(2) * walls(19) +
                    walls(25) * 13) mod 40) +
                    10;
            else
                blocks(31) <= blocks(31)+ ((
                    walls(2) * walls(19) +
                    walls(25) * 13) mod 40) -
                    10;
            end if;
        end if;
        if generate_block = '0' then
            blocks(31) <= 0;
        end if;
    end process;
```

E. image drawing—-worked by Wang

The image drawing is divided into six parts: helicopter, background, walls, obstacles, score and game over screen.

1) drawing helicopter and background:

- Use a 16x23 two-dimensional array to store the basic shape of the helicopter.
- Set the initial position of the helicopter, and calculate the relative position of the pixel block within the helicopter by subtracting the left and top positions of the helicopter from pixel_x and pixel_y respectively.
- If the pixel position is within the helicopter and the relative position of the pixel block is ”1”, draw white (helicopter color); otherwise, draw blue (background color).

2) drawing walls and obstacles:

- **Wall drawing**:
  - Divide the horizontal direction of the wall into 32 regions and use for-loop structure to draw each region. When the vertical pixel position is less than the wall height or greater than the wall height plus the cave width, draw the wall color.
- **Obstacle drawing**:
  - Similar to the principle of wall drawing, when the vertical pixel position is greater than the obstacle height and less than the obstacle height plus 50 (the width of the obstacle), draw the obstacle color.

3) drawing score and game over:

- Store the bitmaps of all characters that need to be used in an array. The drawing principle is similar to drawing the helicopter. When a character needs to be used, determine the character to be drawn using the relative pixel position.
- Once the character is determined, fill in the corresponding pixel color at the pixel positions where the character should be placed. Then the score and the game over screen can display in VGA display.

Finally, transfer all RGB data to the RGB channels of the VGA interface on the board. The image drawing is completed.

V. Pmod: Ultrasonic module—-worked by Kong

The HC-SR04 module is a commonly used ultrasonic distance measurement sensor. It calculates the distance to an obstacle by emitting ultrasonic waves and measuring the echo time. This module features high measurement accuracy, fast response speed, and strong anti-interference capabilities. It is suitable for various distance measurement applications such as robot obstacle avoidance, object detection, and rangefinders.

Basic Working Principle:

1) Use IO port TRIG to trigger distance measurement, providing at least a 10µs high-level signal.
2) The module automatically sends 8 cycles of 40kHz square waves and automatically detects if there is a returned signal;
3) If there is a returned signal, the IO port ECHO outputs a high-level signal. The duration of the high-level signal is the time from when the ultrasonic wave is transmitted to when it is received. The distance is calculated in the following equation:

$$distance = \frac{\text{High-level duration} \times \text{Speed of sound}}{2}$$

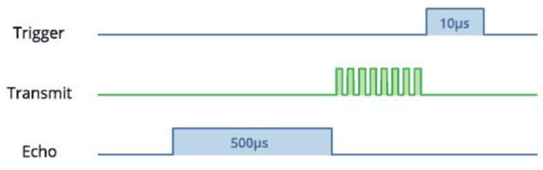


Fig. 3. working signal

This module has four I/O ports as shown below: the input clock CLOCK, the virtual button output BTN that detects whether the hand is close enough to the sensor, and the trigger and echo signals mentioned above.

```
entity ultrasonic is
    port(
    CLOCK: in std_logic;
    BTN: out std_logic;
    TRIG: out std_logic;
    ECHO: in std_logic
    );
end ultrasonic;
```

I first divide the 100 MHz signal on the board to obtain a 1 MHz signal, which has a period of 1 microsecond. By calculating the duration of the high level of the echo signal, the distance from the hand to the sensor is determined. A threshold value, distance, is set, and when it is below this threshold, it is considered as pressing the virtual button, and the BTN outputs a high level; otherwise, it outputs a low level.

The following is the corresponding code:

```
begin

    process(CLOCK)
    variable count0: integer range 0 to 16;
    begin
        if rising_edge(CLOCK) then
            if count0 = 49 then
                count0 := 0;
            else
                count0 := count0 + 1;
            end if;
            if count0 = 0 then
                microseconds <= not
                    microseconds;
            end if;
        end if;
    end process;

    process(microseconds)
    variable count1: integer range 0 to
        262143;
    begin
        if rising_edge(microseconds) then
            if count1 = 0 then
                counter <= "000000000000000000
                    ";
                trigger <= '1';
            elsif count1 = 10 then
                trigger <= '0';
            end if;
            if ECHO = '1' then
                counter <= counter + 1;
            end if;
            if count1 = 249999 then
                count1 := 0;
            else
                count1 := count1 + 1;
            end if;
        end if;
    end process;

    process(ECHO)
    begin
        if falling_edge(ECHO) then
            if counter < distance then
                sigs <= '1';
            else
                sigs <= '0';
            end if;
        end if;
    end process;

    BTN <= sigs;
    TRIG <= trigger;

end arch;
```

1) configurations: The interface of the HC-SR04 is connected to the IO port named JA on the board, and the following image shows the connection method:

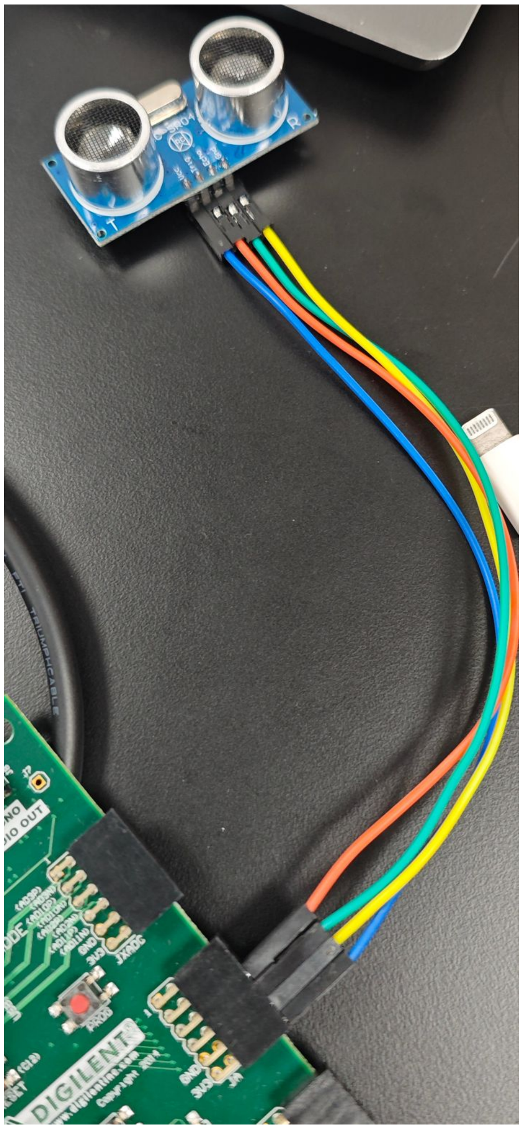


Fig. 4. connection

```
    );
end vga_sync;
```

VI. Pmod: VGA display module—-worked by Wang

VGA uses a raster scanning method to display images. The electron gun in CRT monitors (or pixel control in modern displays) scans from the top left to the bottom right of the screen, line by line. The hsync signal causes the beam to move to the start of the next line, and the vsync signal causes the beam to return to the top of the screen for the next frame.

As the electron gun scans quickly across the screen, the human eye perceives a complete image. The fast scanning rate and synchronization signals work together to create a stable and coherent picture.

A. *Scan period*

- Each clock cycle transmits one pixel of information, and the clock cycle is the working clock of the VGA display.
- Each horizontal scan period corresponds to scanning one row (horizontal line) of the screen.
- Each vertical scan period corresponds to scanning the entire screen (vertical column).
- As the following figure, an image is produced only when both the horizontal and vertical scan periods are within the effective image timing portion.

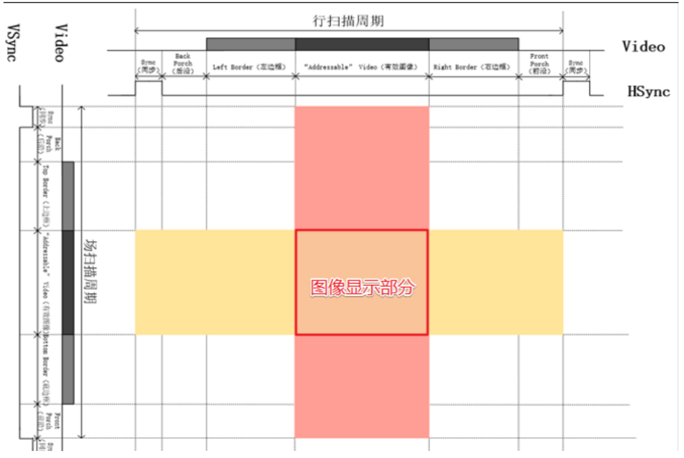


Fig. 5. VGA image generation

B. *Clock Calculation*

We use the 640x480@60 mode, with a pixel clock frequency Vga_clk = $800 \times 525 \times 60 = 25200000 \approx 25.175$ MHz

Therefore, we need a frequency divider to obtain a 25 MHz signal.(the error is negligible).

C. *I/O ports*

```
entity vga_sync is
   port(
      clk, reset: in std_logic;
      btn: in std_logic;
      playAgain: in std_logic;
      hsync, vsync: out std_logic;
      video_on, p_tick: out std_logic;
      pixel_x, pixel_y: out std_logic_vector
          (9 downto 0)
```

Input signals:
- clk: system clock;
- reset: asynchronous reset signal for system initialization;
- btn: the virtual button to play game
- playAgain: the button to play again
Output signals:

- hsync, vsync: the hsync and vsync signals ensure that the pixels are displayed in the correct positions. The hsync signal indicates the end of a row of pixels, while the vsync signal indicates the end of a frame.

- video_on: indicates whether the current pixel is within the visible area of the screen.It is set to '1' when the horizontal and vertical counters are within the display area dimensions (HD and VD, respectively), meaning the pixel is within the active display region.

- p_tick: a pixel clock tick, generated at 25 MHz.

- pixel_x, pixel_y: represents the current horizontal and vertical position of the pixel being processed.

The overall logic of the code is to first define all necessary constants, then use a frequency divider to obtain the required clock, and subsequently generate the `hsync` and `vsync` signals to ensure proper pixel generation. Finally, when the current pixel is within the display area, the `video_on` signal is set to 1.

The following is the corresponding code:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity vga_sync is
   port(
      clk, reset: in std_logic;
      btn: in std_logic;
      playAgain: in std_logic;
      hsync, vsync: out std_logic;
      video_on, p_tick: out std_logic;
      pixel_x, pixel_y: out std_logic_vector
          (9 downto 0)
    );
end vga_sync;

architecture arch of vga_sync is
   -- VGA 640-by-480 sync parameters
   constant HD: integer:=640; --horizontal
       display area
   constant HB: integer:=48 ; --h. back porch
   constant HF: integer:=16 ; --h. front porch
   constant HR: integer:=96 ; --h. retrace
   constant VD: integer:=480; --vertical
       display area
   constant VB: integer:=33;  --v. back porch
   constant VF: integer:=10;  --v. front porch
   constant VR: integer:=2;   --v. retrace

   -- clock frequency divider
   signal clk_divider : unsigned(1 downto 0)
       := "00";
   -- sync counters
   signal v_count_reg, v_count_next: unsigned
       (9 downto 0);
   signal h_count_reg, h_count_next: unsigned
       (9 downto 0);

   -- output buffer
   signal v_sync_reg, h_sync_reg: std_logic;
   signal v_sync_next, h_sync_next: std_logic;

   -- status signal
   signal h_end, v_end, pixel_tick, clk50mhz,
       clk25mhz : std_logic;
begin

   -- clock divider to divide by 4
   p_clk_divider: process(reset, clk,
       clk_divider)
   begin
     if(reset='1') then
       clk_divider <= (others=>'0');
     elsif(rising_edge(clk)) then
       clk_divider <= clk_divider + 1;
     end if;
   end process p_clk_divider;

   -- 50 and 25 MHz clocks
   clk50mhz <= clk_divider(0);
   clk25mhz <= clk_divider(1);

   -- 25 MHz pixel tick
   pixel_tick <= clk25mhz;

   -- registers
   process(clk50mhz, reset)
   begin
       if reset='1' then
          v_count_reg <= (others=>'0');
          h_count_reg <= (others=>'0');
          v_sync_reg <= '0';
          h_sync_reg <= '0';
       elsif (clk50mhz'event and clk50mhz='1')
            then
          v_count_reg <= v_count_next;
          h_count_reg <= h_count_next;
          v_sync_reg <= v_sync_next;
          h_sync_reg <= h_sync_next;
       end if;
   end process;

   -- status
   h_end <=  -- end of horizontal counter
      '1' when h_count_reg=(HD+HF+HB+HR-1)
          else --799
      '0';
   v_end <=  -- end of vertical counter
      '1' when v_count_reg=(VD+VF+VB+VR-1)
          else --524
      '0';

   -- mod-800 horizontal sync counter
   process (h_count_reg,h_end,pixel_tick)
   begin
      if pixel_tick = '1' then  -- 25 MHz tick
         if h_end='1' then
            h_count_next <= (others=>'0');
         else
            h_count_next <= h_count_reg + 1;
         end if;
      else
         h_count_next <= h_count_reg;
      end if;
   end process;

   -- mod-525 vertical sync counter
   process (v_count_reg,h_end,v_end,pixel_tick
       )
   begin
      if pixel_tick = '1' and h_end='1' then
         if (v_end='1') then
            v_count_next <= (others=>'0');
         else
            v_count_next <= v_count_reg + 1;
         end if;
      else
         v_count_next <= v_count_reg;
      end if;
   end process;

   -- horizontal and vertical sync, buffered
       to avoid glitch
   h_sync_next <=
      '1' when (h_count_reg>=(HD+HF))
                  --656
           and (h_count_reg<=(HD+HF+HR-1))
               else --751
      '0';

   v_sync_next <=
      '1' when (v_count_reg>=(VD+VF))
                  --490
           and (v_count_reg<=(VD+VF+VR-1))
               else --491
      '0';

   -- video on/off
   video_on <=
      '1' when (h_count_reg<HD) and (
          v_count_reg<VD) else
      '0';

   -- output signal
   hsync <= h_sync_reg;
   vsync <= v_sync_reg;
   pixel_x <= std_logic_vector(h_count_reg);
   pixel_y <= std_logic_vector(v_count_reg);
   p_tick <= pixel_tick;

end arch;
```

VII. Schedule and Budget

A. schedule

- Week 10: Purchase the necessary modules for the project
- Week 11: Draw the program flowchart
- Weeks 12-13: Learn relevant knowledge and implement basic functions
- Weeks 14-15: Conduct testing and improvements
- Week 16: Project presentation and report submission

B. budget

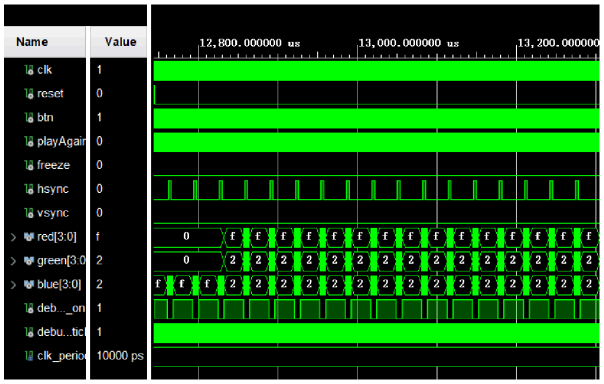
- VGA display 138CNY

- HC-SR04 ultrasonic module 5CNY

VIII. Conclusion

A. *result analysis –worked by Wang*

The following is the initial simulation result:



Name	Value
clk	1
reset	0
btn	1
playAgair	0
freeze	0
hsync	0
vsync	0
red[3:0]	f
green[3:0	2
blue[3:0]	2
deb..._on	1
debu...tic	1
clk_perio	10000 ps

Fig. 6. simulation result

Then I performed a critical path analysis, and it can be seen that the maximum delay path is likely when generating the walls.

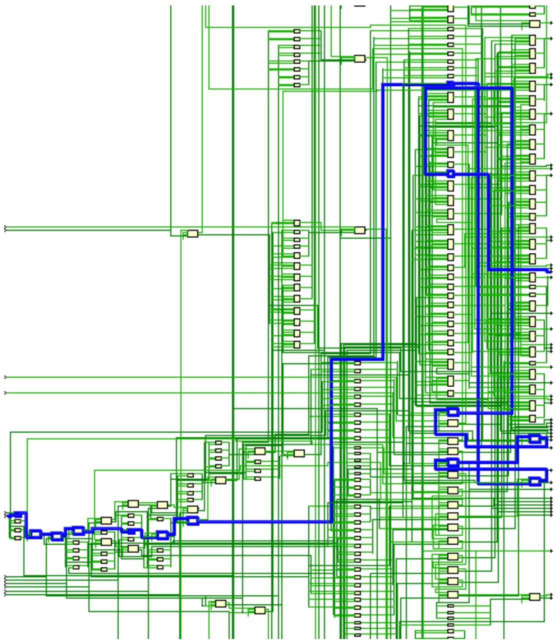
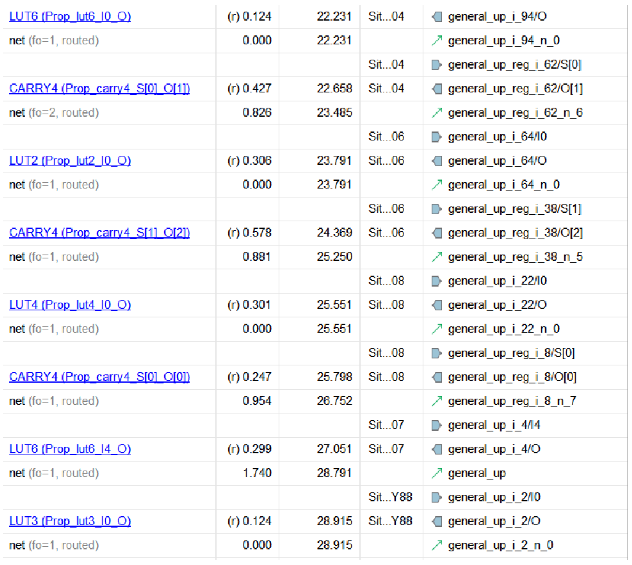


Fig. 7. critical path



LUT6 (Prop_lut6_I0_O)	(r) 0.124	22.231	Sit...04	general_up_i_94/O
net (fo=1, routed)	0.000	22.231		general_up_i_94_n_0
			Sit...04	general_up_reg_i_62/S[0]
CARRY4 (Prop_carry4_S[0]_O[1])	(r) 0.427	22.658	Sit...04	general_up_reg_i_62/O[1]
net (fo=2, routed)	0.826	23.485		general_up_reg_i_62_n_6
			Sit...06	general_up_i_64/I0
LUT2 (Prop_lut2_I0_O)	(r) 0.306	23.791	Sit...06	general_up_i_64/O
net (fo=1, routed)	0.000	23.791		general_up_i_64_n_0
			Sit...06	general_up_reg_i_38/S[1]
CARRY4 (Prop_carry4_S[1]_O[2])	(r) 0.578	24.369	Sit...06	general_up_reg_i_38/O[2]
net (fo=1, routed)	0.881	25.250		general_up_reg_i_38_n_5
			Sit...08	general_up_i_22/I0
LUT4 (Prop_lut4_I0_O)	(r) 0.301	25.551	Sit...08	general_up_i_22/O
net (fo=1, routed)	0.000	25.551		general_up_i_22_n_0
			Sit...08	general_up_reg_i_8/S[0]
CARRY4 (Prop_carry4_S[0]_O[0])	(r) 0.247	25.798	Sit...08	general_up_reg_i_8/O[0]
net (fo=1, routed)	0.954	26.752		general_up_reg_i_8_n_7
			Sit...07	general_up_i_4/I4
LUT6 (Prop_lut6_I4_O)	(r) 0.299	27.051	Sit...07	general_up_i_4/O
net (fo=1, routed)	1.740	28.791		general_up
			Sit...Y88	general_up_i_2/I0
LUT3 (Prop_lut3_I0_O)	(r) 0.124	28.915	Sit...Y88	general_up_i_2/O
net (fo=1, routed)	0.000	28.915		general_up_i_2_n_0

Fig. 8. critical path analysis

B. *conclusion and limitation*

The following is the picture of our real product:

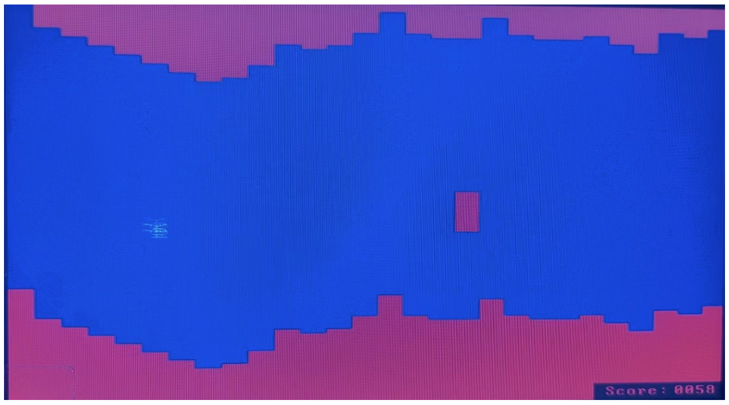


Fig. 9. real product

We have achieved most of our expected goals, and the game is fully playable, offering both fun and challenges. However, due to time constraints, our project is not yet perfect. For example, we cannot freely control the speed and difficulty, nor can we record previous scores. Overall, this has been a very rewarding project experience. From designing the game to debugging it, this project has enhanced our ability to write VHDL code and provided new insights into FPGA development that we might encounter in future work.