Digital Logic Project

Magic Tower Design Report

Zhang Chengyi 3150104031

Lu Jikai 3150102220

Wen Zihao 3150101213

Wang Daxin 3150103559

Date: 2017-1-7

Content

Section 1: Introduction

1.1 Main function of the whole circuit

We designed a very classic game called Magic tower.

In this game, you can press the keys of up, down, left and right in the keyboard to control the movement of the character and press the number keys in the main area of the keyboard to make purchase, if possible. You can gain money and experience through defeating monsters on the map, and increase blood volume, enhance attack power or defense power of the character using these to make purchase. Every map is a maze, so you must found keys with three different colors to open those doors. When you become strong enough, you can conduct final battle with the final boss and rescue your princess.

Also, we have design a background music imitating the original playing with the buzzer on the Arduino. Besides, if you cannot continue to play this game because you have taken some wrong steps, you can just toggle the switches on the board to turn on cheat mode to clear maps fast.

1.2 Why we chose to design this project

Magic tower was one of the most popular flash games at our childhood, and even today it is still well-known in the Internet for being easy to play and full of variability. Actually, there are some templates that can produce different types of magic tower games spreading through the Internet. So it is very easy to design a game like the game we design running on the computer.

However, we cannot found anyone who has done magic tower based on FPGA. Thinking of the large number of materials of this game that can be found in the Internet and to challenge ourselves to finish something others have not done, we finally chose to design the game Magic tower and fortunately we finally finished it before encountering insurmountable problems.

Section 2: Design specification

2.1 Development environment

Experiment platform

Family: Kintex7
Device: XC7K160T
Package: FFG676

Development environment

Xilinx ISE 14.7

HDL

Verilog

2.2 Requirement of the project

This game needs three hardware parts to play, that are a development board as shown in section 2.1, a monitor with VGA and a keyboard with USB.

Input

SW [15:0]: the sixteen switches on the board

keyboard: use up, down, left and right to control the movement of the character and use the number key on the main area of the keyboard to make a purchase.

Output

Monitor: to display the game screen.

Buzzer: to play the background music

7-segment digital: to show the key value of the keyboard button pressed.

2.3 How to play our Magic Tower

2.3.1 Prepare for the game

Firstly, you need to connect a keyboard with USB and a monitor with VGA to the development board. After programming the bit stream file successfully, you can find that the game has begun with the first floor map displayed on the monitor.

2.3.2 Rule of the game

Control

Use ↑(up), ↓(down), ←(left), →(right) to control the movement of the hero, the main character of this game.

Battle

Beat the creatures in the tower and get money and experience! Walk directly towards them and a fight will automatically start. The state of enemies, including HP, attack and defend are displayed on the lower right corner of the screen. The hero and his enemies will attack in turns. The damage per turn is equal to the attacker's attack minus the defender's defend. If the attacker's attack is smaller than the defender's defend, then The damage per turn is 0. When the HP of enemies comes to 0 or less, the fight stops and the enemies will disappear from the map. If the HP of the hero is lower than the damage caused by enemies, a withdrawal will happen, that is the hero will withdraw and the enemies will still remain on the map.

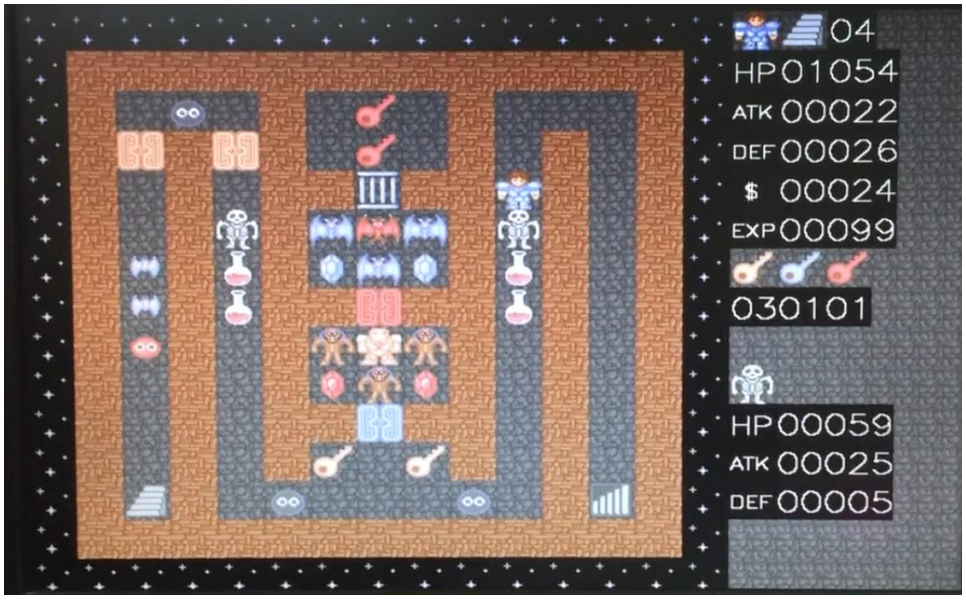


Figure 1 Main interface

Door

Three kinds of doors are in the tower: yellow doors, blue doors and red doors. The door won't open if you don't have any keys of the same color. The number of different kinds of keys are displayed on the right hand side. There are two ways to get keys. One is to collect them in the tower, the other is to buy them from the merchants on 2nd floor, 5th floor, 12th floor and 15th floor. An interface will appear on the lower right corner of the screen when walking up to the merchants. Press 1 to pay $25 for a yellow key, 2 to pay $50 for a blue one and 3 to pay $100 for a red one. Keys are valuable, so don't waste them to open the gates that are not necessary to open until you are sure that you have enough keys or enough money to buy keys!

Props

Several common items in the tower can strengthen the ability of our hero: red potion (+200HP), blue potion (+500HP), red gem (+3 attack) and blue gem (+3 defend).

A sword and 2 shields are in the tower: iron sword (+10 attack, on 3rd floor), iron shield (+10 defend, on 5th floor) and large shield (+60 defend, on 11th floor). There are also some rare and powerful treasures waiting for you to explore!

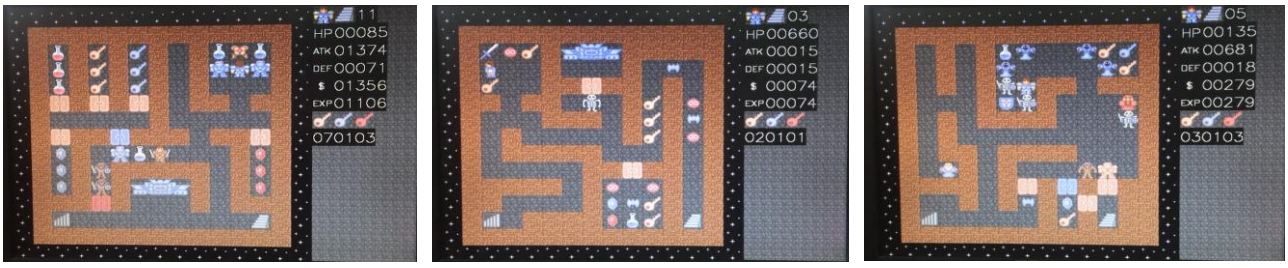


Figure 2 Props

Purchase

You can pay $25 for adding 4 attack or 4 defend or 500 HP for the hero in the shops on both 3rd floor and 11th floor. To enter the shop, you should first walk up to the "head" of the shop and then press ↑(up). A shop interface will appear on the lower right corner of the screen. Press 1 for 500 HP, 2 for 4

attack, 3 for 4 defend. If you don't have enough money or you don't want to buy any of the above, press space to quit the shop.

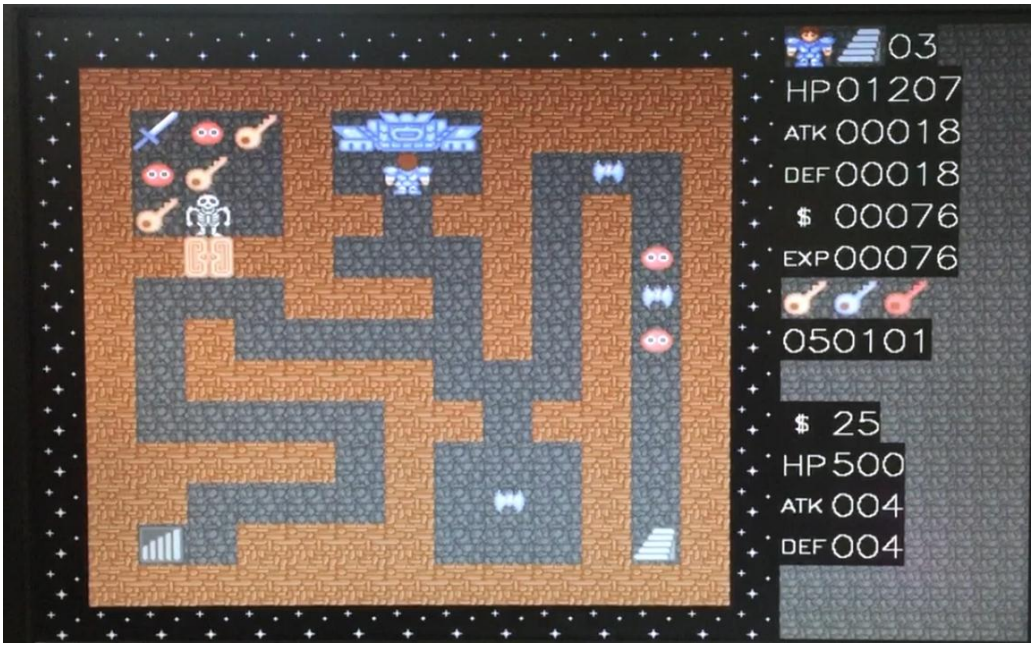


Figure 3 Money Shop

You can also use experience to strength the hero. On 2nd floor, 5th floor, 13th floor and 15th floor there are 3 old man. Walk up to them and an interface will on the lower right corner of the screen. Press 1 to pay 30 exp for 1000 HP, 2 for 7 attack and 3 for 7 defend. Press space to quit the interface.

Figure 4 Exp Shop

Cheating Mode

A cheat system is established in order to make our test easier. Set SW[1] (or SW[2]) to 1 and the hero's HP (or attack) will increase every move.

Section 3: Model Design

3.1 Model relationship

In this game, we have the attributes of the character and monsters and keys' number displayed on the docker, have map displayed on the main area of the monitor. To deal with these data and display, we choose to use MVVM architecture to design this game. MVVM architecture has three parts: Model, View and ViewModel. As shown below.

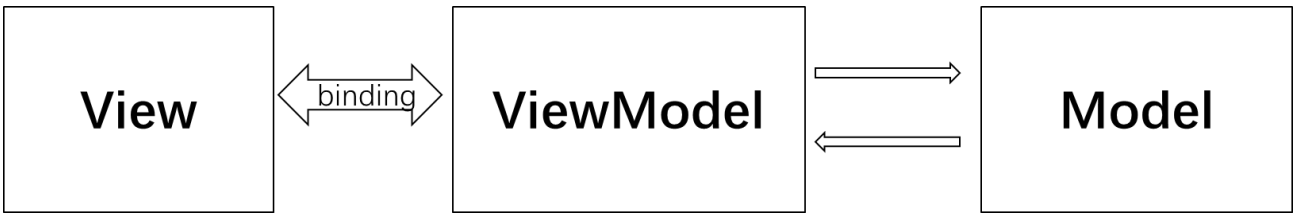


Figure 5 MVVM structure

Model refers either to a domain model, which represents real state content (an object-oriented approach), or to the data access layer, which represents content (a data-centric approach). In our game, the main logic is like this part.

View is the structure, layout, and appearance of what a user sees on the screen. The VGA module can gain data from View model and display the appearance on the monitor.

View model is an abstraction of the view exposing public properties and commands. MVVM has a binder between View and View model. In the view model, the binder mediates communication between the view and the data binder. The view model has been described as a state of the data in the model. This part deals with the display logic.

And another feature of the project is reactive programming, there are many variables in the code that can change with others synchronously. So it is easy to express static or dynamic data flow, and the associated calculation model will automatically spread the value of the data flow changing.

3.2 Top model

This is one of the most important parts of our project, and the main logic of the game is in this part which includes memory module, input module and display module.

Input

clk,
SW[15:0]
ps2_clk,
ps2_data

Output

HSYNC,
VSYNC,

Red[3:0]
Green[3:0]
Blue[3:0]
Buzzer
LED[7:0]
AN[3:0]
SEGMENT[7:0]

Function

To control the game to run as expected.

The flow chart of the top is shown below.

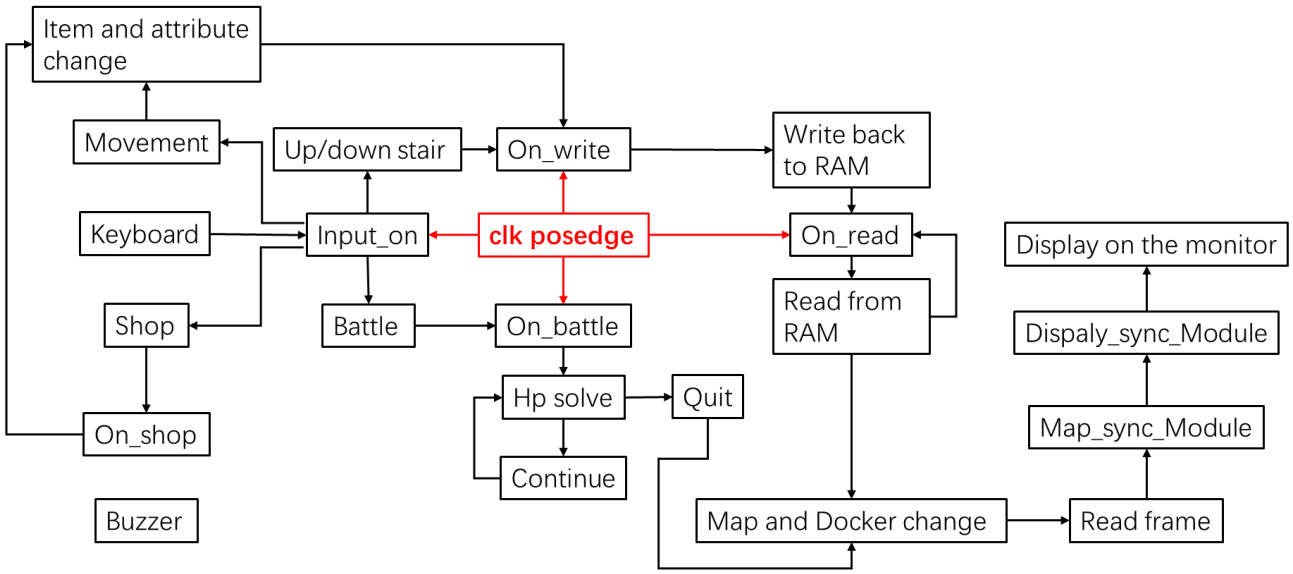


Figure 6 The flow chart of Top

So as you can see, at every posedge of clk, the program will check four states: input, battle, write and read. If the state of read is true, the program will read something called from the RAM and display that on the monitor. If the state of write is true, it will write the latest state into the RAM and read it at the next true state of read. If the state of battle is true, it will begin to solve the hp changes until one of the termination conditions is triggered, such as the monitor being defeated. After winning the battle, the monster will disappear from the map and the change will be displayed on the monitor.

If the states of write, read and battle are all false and the state of input is true, then you can control the character's movement using keyboard. After the user presses the keyboard with up, down, left or right, there will be another four states to judge: up/down stair, battle, shop or movement. If the character touches the stair, it will set the write state as true and read the next floor map from RAM, and then display it on the monitor. If he touches a monster, the program will set the state of battle true and conduct battle. If he touch a shop, it will trigger the shop state and the player can use the keys of 1,2,3 to make purchase and when the attributes of the character changes, the result will be displayed on the docker. Besides, when the character has no states shown as above, it will just chang the position of the character and display this change on the map.

3.3 VGA model

Our Magic Tower uses VGA display module. We have two modules here: one is vga_display, and the other is vga_sync.

3.3.1 vga_display module

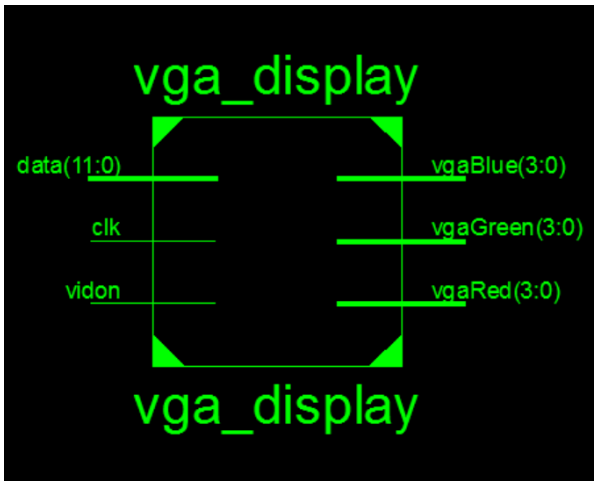


Figure 7 vga_display RTL diagram

Input

clk: 25MHz

vidon: vidon = 1 when it is in display mode, vidon = 0 when it is not

data: RGB value with 12 bits

Output

vgaBlue value (4 bits)

vgaGreen value (4 bits)

vgaRed value (4 bits)

Function

The VGA need 5 signals, three color values and two synchronizing signal. This part is used to check whether it is in the display mode. If it is, then we give the RGB value to the monitor, else we will set all the color value to be 0.

We can use pseudocode as follows to describe the workflow of it.

```
module vga_display(
                input wire clk,
                input wire vidon,
                input wire [11:0] data,
                output reg [3:0] vgaRed,
                output reg [3:0] vgaGreen,
                output reg [3:0] vgaBlue
);
always @(posedge clk)
      begin
            if (vidon == 1)
            begin
                  copy data to vgaRed, vgaGreen, vgaBlue;
            end
            else
            begin
                  set vgaRed, vgaGreen, vgaBlue to be 0;
            end
      end
endmodule
```

3.3.2 vga_sync module

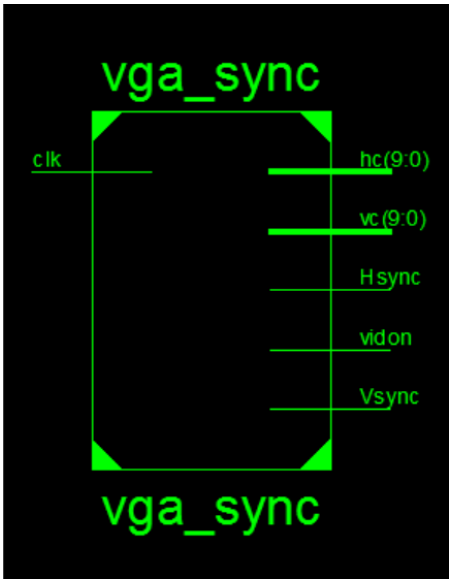


Figure 8 vga_sync RTL diagram

Input

clk: 25MHz

Output

hc: the horizontal scanning signal

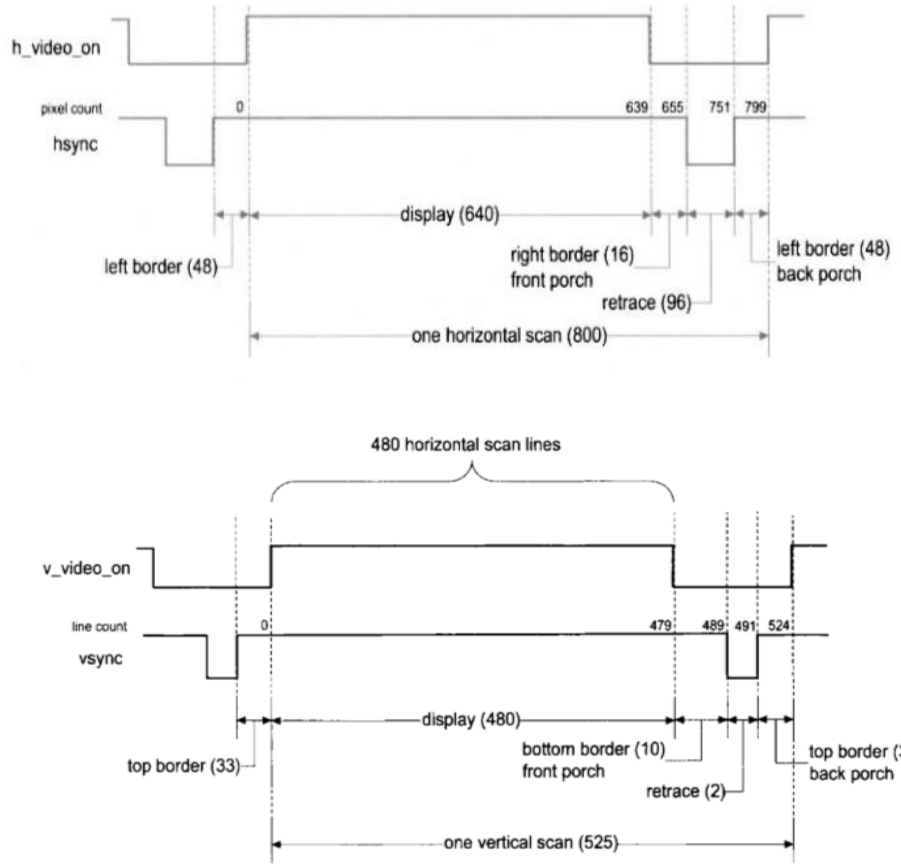
vc: the vertical scanning signal

Hsync: the horizontal syncronizing signal

Vsync: the vertical syncronizing signal

vidon: the display signal

Function



Above is the timing diagram for horizontal scan and vertical scan. When a clk signal comes, we add hc by 1. If one row has been completed, we add vc by 1, and reset hc to be 0. Thus we can scan the monitor row by row to display the picture. At the same time, we should check the synchronizing signal. If the hc and vc are in the display line or boarder, we should set synchronizing signal to be 1. And if the hc and vc are in display line, we set vidon to be 1.

Next we will use pseudocode to describe how this part woks.

```
module vga_sync(
        input wire clk,
        output reg Hsync,
        output reg Vsync,
        output reg[9:0] hc,
        output reg[9:0] vc,
        output reg vidon
);

    always @(posedge clk)
    begin
        if (hc reaches the end)
        begin
```

```
            set hc to be 0;
        end
        else
        begin
            hc++;
        end
    end

    always @(posedge clk)
    begin
        if (hc has reached the end)
        begin
            if (vc reaches the end)
            begin
                set vc to be 0;
            end
            else
            begin
                vc++;
            end
    end

    always @(*)
    begin
        if (hc in retrace)
            Hsync = 0;
        else
            Hsync = 1;
    end

    always @(*)
    begin
        if (hc in retrace)
            Vsync = 0;
        else
            Vsync = 1;
    end

    always @(*)
    begin
        if (hc and vc are in display line)
        begin
            vidon = 1;
        end
```

```
            else
                vidon = 0;
        end
endmodule
```

3.3.3 The display of map

The map consists of 225(15*15) blocks with 32*32 pixels. First we read in all the RGB values in bitstream for all 32*32 figures we want to display. Then we give each of the figure with an identified number. So the map is a 2-dimention array which stores all the identified numbers of 225 figures. Then the people who writes logic only need to operate on the map array. Given an array map, what we need to do is display the map on monitor. The pseudocode is shown below:

```
always @( posedge clk ) begin
        if ( vidon == 0 ) begin
                data = 0;
        end else begin
            if(display is in map area)begin
                data = bitstream(figure ID, nowx, nowy);
// if we know the figure ID and the x, y value, we can know what color value should be in the pixel now.
            end
        end
end
```

3.3.4 The display of side bar

The principle for side bar is similar to that of map. The side bar contains 150 blocks with 16*32 pixels. We must notice that this time we only display half of the figure from the bitstream.
The pseudocode is shown below:

```
always @( posedge clk ) begin
        if ( vidon == 0 ) begin
                data <= 12'b0;
        end else begin
            if(display in sidebar area )begin
                data <= bitstream(figure ID,left_or_right,nowx,nowy);
//except for the ID and x, y value, we have to know whether we want to display left part or the right part.
            end
        end
end
```

3.4 Keyboard model

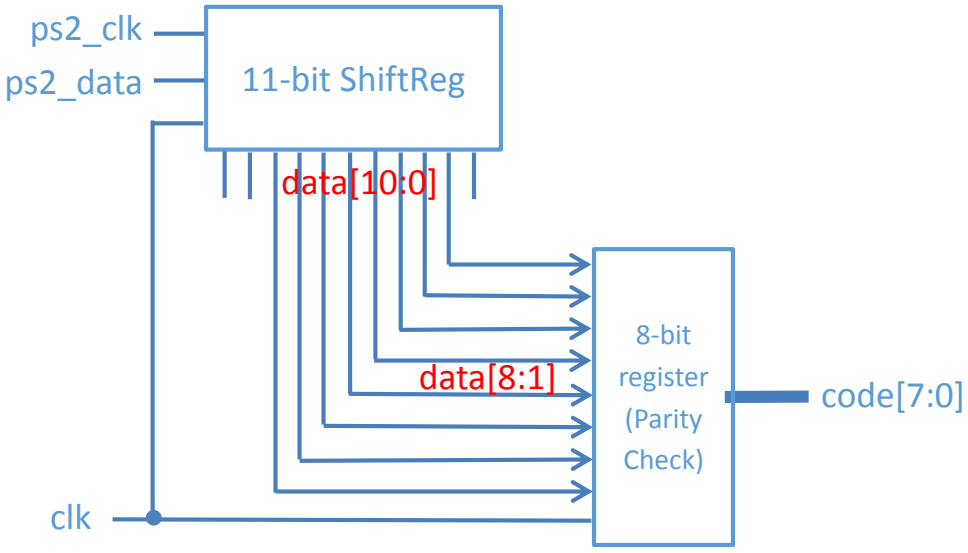


Figure 9 Keyboard ports

Input

Clk: 100MHz
ps2_clk
ps2_data

Output

AN[3:0]
SEGMENT[7:0]
code[7:0]

Function

Read input keys from keyboard and output its ps/2 code in code[7:0].

To display the code[7:0], we use 7-segment display which has been used many times in the class, so here we don't repeat the principle of it but just give the brief information.

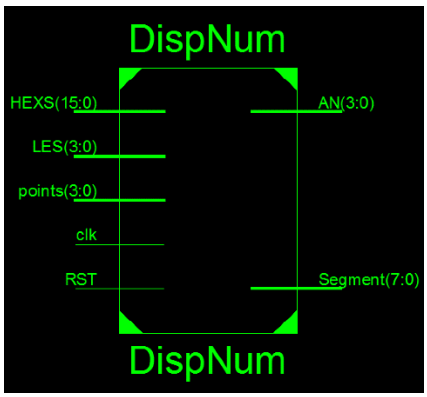


Figure 10 DispNum RTL diagram

Input

HEXS[15:0]
LES[3:0]
points[3:0]
clk
RST

Output

AN[3:0]
Segment[7:0]

Function

According to the data of input, display numbers using 7-segment digital.

How does the ps/2 keyboard work?

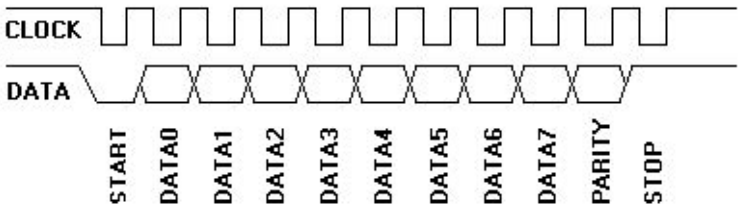


Figure 11 The timing diagram of ps/2

There are two output signals from the keyboard. One is the clock signal ps2_clk, the other is the keyboard code ps2_data. The signal ps2_clk and ps2_data are both 1 when there aren't any inputs from the keyboard. When someone press a key on the keyboard, ps2_clk and ps2_data will behave like the figure above. An 11-bit signal will be transferred one bit by one bit through ps2_data. The first bit is start bit, which is 0 all the time. The last bit is stop bit, which is 1. The 9-bit in the middle is the 8-bit key code and a parity bit (even parity).

To illustrate the workflow more clearly, the pseudocode of how to identify the buttons in keyboard is shown as follows.

```
always @(posedge clk/250) begin
        if (negedge ps2_clk) begin
                if (count==11||timing>=MaxTiming) begin
                        cnt <= 0;
                        if (pass parity check) code<=data[8:1];
                end else if (cnt < 11) begin
                        data[10:0] <= {ps2_data,data[10:1]};
                        cnt <= cnt + 1;
                end
        end
end
```

And the diagram below shows the numbers corresponding to the keyboard buttons which is in the lab.

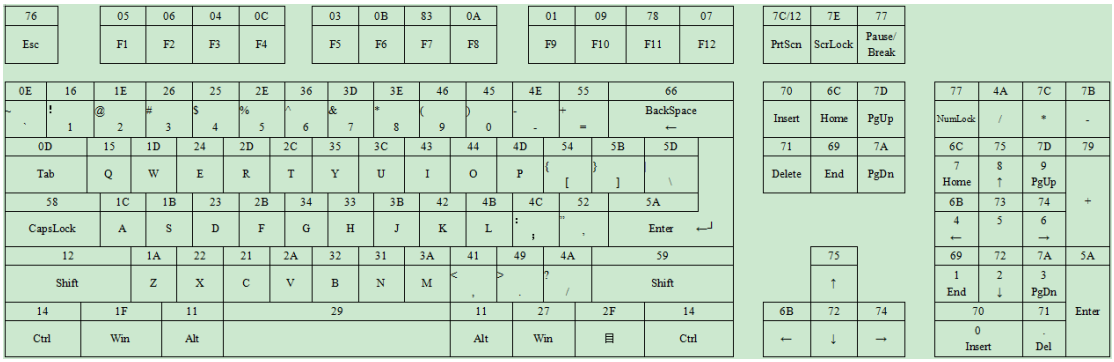


Figure 12 Keyboard keys

Finally, we give the source code with comments of the keyboard module.

```
module Keyboard(
        input clk, ps2_clk, ps2_data,
        output wire [3:0] AN,
        output wire [7:0] SEGMENT,
        output reg [7:0] code
        );

        reg reading;                            //this is 1 if still waits to receive more bits
        reg [11:0] timing;                      //store the time passed since it received the last bit
        reg last_clk;                           //store the previous state of ps2_clk
        reg [10:0] data;                        //store the 11 received bits
        reg [3:0] cnt;                          //tells how many bits were received (from
0 to 11)
        reg trigger = 0;                        //acts as a 250 times slower clock than clk
        reg [7:0] down_counter = 0;//a counter counts from 0 to 249

        //Set initial values
        initial begin
                last_clk = 1;
                data = 0;
                cnt = 0;
                reading = 0;
                timing = 0;
code = 0;
        end

        always @(posedge clk) begin                     //reduce the clk's frequency 250
times
                if (down_counter < 249) begin           //use variable trigger as the new clock
                        down_counter <= down_counter + 1;
```

```
                        trigger <= 0;
                end
                else begin
                        down_counter <= 0;
                        trigger <= 1;
                end
        end

        always @(posedge clk) begin
                if (trigger) begin
                        if (reading)                            //if it still waits to read
full packet of 11 bits, then
                                timing <= timing + 1;           //count up timing
                        else
                                timing <= 0;                    //if it stops reading,
then reset timing
                end
        end
        always @(posedge clk) begin
                if (trigger) begin                              //if the down counter (clk/250) is
ready
                        if (ps2_clk != last_clk) begin          //if the state of keyboard clock
changes
                                if (!ps2_clk) begin             //and if the ps2_clk is at negative
edge
                                        reading <= 1;           //mark down that it is
still reading for the next bit
                                        data[10:0] <= {ps2_data,data[10:1]};
//add up the data received by shifting bits and adding one new bit
                                        cnt <= cnt + 1;         //the number of bit
received increases 1
                                end
                        end
                        else if (cnt == 11) begin               //if it already received 11 bits
                                cnt <= 0;                       //reset the number of
bits received
                                reading <= 0;                   //mark down that
reading stopped
                        end
                        else begin                              //if it hasn't received
full pack of 11 bits
                                if (cnt < 11 && timing >= 4000) begin
//and if after a certain time no more bits were received, then
                                        cnt <= 0;               //reset the number of
```

```
bits received
                                                reading <= 0;                   //and wait for the next
packet
                                            end
                                        end
                                last_clk <= ps2_clk;                            //mark down the previous state of
the keyboard clock
                                end
                        end

always @(negedge reading) code<=data[8:1];
//read the data received only when finish reading

                        disp_num mm(clk,{8'b0,code[7:0]}, 4'b0, 4'b0, 1'b0, AN, SEGMENT);
//check received data
endmodule
```

3.5 Buzzer model

We use the buzzer in the Arduino to play the background music of magic tower and the actual effect of it is shown in the video of describing our project. Limited by the structure of the buzzer, we can just play a music using monotonic sound.

To play music with a buzzer, we must be able to control the tone of it. According to the working principle of buzzer, we can generate square waves with different frequency (PWM waves) to make the buzzer let out different tones in according with the frequency. So the main task is to design a module which can generate PWM waves.

To generate PWM waves, there needs a counter to divide the frequency. But for there is a 100MHz clock provided by the board which is too high, the counter will be very large if we just use it to divide the frequency, so we first divided the 100MHz clock to 50MHz.

As shown above, the buzzer model has three parts: clk_50MHz module, pwm_generator module to generate PWM waves with different frequency and buzzer module to generate different tones and store music notes.

3.5.1 clk_50MHz module

Input
clk: 100MHz
Output
clk_50M
Function

Divide the 100MHz clock to 50MHz so that the counter becomes smaller when we generate PWM waves.
To implement this module, we can just use 1 bit as a counter. When the clk is at its positive edge, the counter adds itself by 1 and for there are only 0 and 1 as the values of the counter, we can just make clk_50M be equal to it. And in this way we get a clock with 50MHz.

```
module clk_50(
    input   clk,
    output clk_50M);
        reg cnt = 1'b0;
        always @ (posedge clk) begin
                if(cnt == 1'b1)
                        cnt <= 1'b0;
                else
                        cnt <= 1'b1;
        end
    assign clk_50M = cnt;
endmodule
```

3.5.2 pwm_generator module

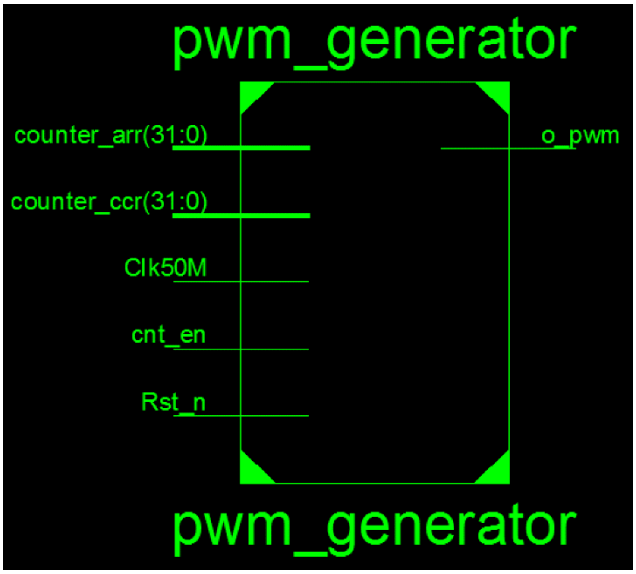


Figure 13 pwm_generator RTL diagram

Input

counter_arr[31:0]: the counter value which is corresponding with the specific tone frequency.
counter_ccr[31:0]: half of the value of counter_arr which is to generate PWM waves with 50% duty cycle
clk50M: clock with 50MHz
cnt_en: enable signal
rst_n: reset signal

Output

o_pwm: when it is 0, the buzzer will beep and 1 not.

Function

This module is to generate the PWM waves to make the buzzer let out different frequency tones.

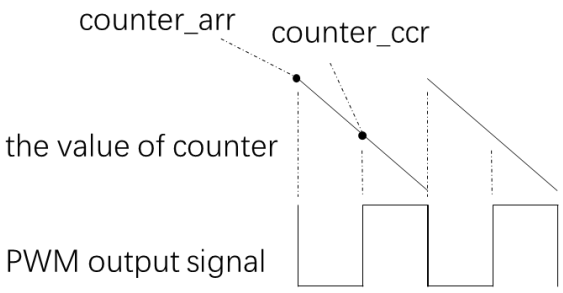


Figure 14 Priciple

Firstly, the value of counter is equal to counter_arr, and after that at every positive edge of clk, counter will subtract 1 from itself. When counter is larger than counter_crr, the output signal will be 0 which means the buzzer will beep and when counter is smaller than counter_crr, the output signal will be 1 which mean the buzzer will not beep.

So different values of counter_arr can define different frequency tones. The corresponding relationship can be shown by the blow equation.

$$f_{pwm} = \frac{clk}{counter\_arr + 1}$$

Reading the pseudocode can understand this progress.

```
module pwm_generator(Clk50M,
                     Rst_n,
                      cnt_en,
                     counter_arr,
                      counter_ccr,
                     o_pwm );
    input Clk50M;     //50MHz
    input Rst_n;      //reset
    input cnt_en;     //enable
          input [31:0]counter_arr;// the counter value which is corresponding with the specific tone frequency.
    input [31:0]counter_ccr;// different values can generate waves with different duty cycle
    output reg o_pwm;     //
    reg [31:0]counter;//
    always@(posedge Clk50M or negedge Rst_n)
                    if(press the reset button)
                                counter <= 32'd0; //reset
                    else if(the enable signal is 1)
                    begin
                                if(counter == 0)
                counter <= counter_arr;
                                else
                counter <= counter - 1'b1;
```

```
                end
                else // the enable signal is 0
                        counter <= counter_arr;
        always@(posedge Clk50M or negedge Rst_n)
                if(press the reset button)
                        buzzer doesn't beep
                else if(counter >= counter_ccr)
                        the buzzer begins to beep
                else
                        the buzzer doesn't beep
endmodule
```

3.5.3 buzzer module

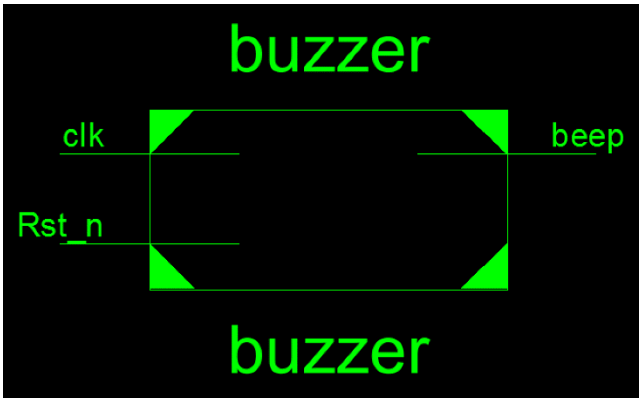


Figure 15 buzzer RTL diagram

Input
clk:100MHz
rst_n: reset signal
Output
beep
Function
To generate different tones and store music notes.

The frequency of PWM waves is just that of tones. So according to the equation

$$f_{pwm} = \frac{clk}{counter\_arr + 1}$$

and the frequency of tones

音名	频率（Hz）	音名	频率（Hz）	音名	频率（Hz）
低音 1	261.6	中音 1	523.3	高音 1	1045.5
低音 2	293.7	中音 2	587.3	高音 2	1174.7
低音 3	329.6	中音 3	659.3	高音 3	1318.5
低音 4	349.2	中音 4	698.5	高音 4	1396.9
低音 5	392	中音 5	784	高音 5	1568
低音 6	440	中音 6	880	高音 6	1760
低音 7	493.9	中音 7	987.8	高音 7	1975.5

```
localparam
    L1 = 191130, //低音1
    L2 = 170241, //低音2
    L3 = 151698, //低音3
    L4 = 143183, //低音4
    L5 = 127550, //低音5
    L6 = 113635, //低音6
    L7 = 101234, //低音7
    M1 = 95546, //中音1
    M2 = 85134, //中音2
    M3 = 75837, //中音3
    M4 = 71581, //中音4
    M5 = 63775, //中音5
    M6 = 56817, //中音6
    M7 = 50617, //中音7
    H1 = 47823, //高音1
    H2 = 42563, //高音2
    H3 = 37921, //高音3
    H4 = 35793, //高音4
    H5 = 31887, //高音5
    H6 = 28408, //高音6
    H7 = 25309, //高音7
    VO = 200000; //空拍
```

Figure 16 The frequency of tones and the values of counter_arr

we can calculate the corresponding value of counter_arr which is shown above.

Every music has “beat”, which is the time period between two notes. We can implement it through dividing frequency. The code is as follows.

```
        //10Hz beat
always @(posedge Clk50M,negedge Rst_n)
        begin
        if(!Rst_n)
                cnt <= 24'd0;
        else
        begin
                cnt <= cnt + 1'b1;
                if(cnt==24'd4999999)
                cnt <= 24'd0;
        end
end
```

We use variable Pitch_num to store the notes of music, every beat we make Pitch_num plus one and every case of Pitch_num we make counter_arr get a value which is the tone we want to play. In this case, there are 97 notes altogether, so when Pitch_num is 97, we give 0 to it so that the buzzer will play the music as a loop.

```
    //notes
always@(posedge Clk50M or negedge Rst_n)
        if(!Rst_n)
            Pitch_num <= 8'd0;
        else if(cnt == 0)
        begin
                if(Pitch_num == 8'd97)
                        Pitch_num <= 8'd0;
```

```
                else
                        Pitch_num <= Pitch_num + 1'd1;
        end
        else
                Pitch_num <= Pitch_num;
```

Except twenty-one tones, there is one vacuum beat whose counter_arr value is 200000. When we give this value, the buzzer will don't beep. The code is as follows.

```
        always@(posedge Clk50M or negedge Rst_n)
        begin
                if(!Rst_n)
                        counter_ccr <= 32'd0;
                else if(counter_arr < 32'd199999)
                        counter_ccr <= counter_arr >> 1;
                else
                        counter_ccr <= 0;
        end
```

Finally, we give the notes of the background music of magic tower here.

```
always@(*)
        case(Pitch_num)
                        0:counter_arr = V0;
                        1:counter_arr = L6;
                        2:counter_arr = V0;
                        3:counter_arr = M1;
                        4:counter_arr = V0;
                        5:counter_arr = M3;
                        6:counter_arr = V0;
                        7:counter_arr = L6;
                        8:counter_arr = V0;
                        9:counter_arr = M1;
                        10:counter_arr = V0;
                        11:counter_arr = M3;
                        12:counter_arr = V0;
                        13:counter_arr = L6;
                        14:counter_arr = V0;
                        15:counter_arr = M1;
                        16:counter_arr = V0;
                        17:counter_arr = M3;
                        18:counter_arr = V0;
                        19:counter_arr = L6;
                        20:counter_arr = V0;
                        21:counter_arr = M1;
                        22:counter_arr = V0;
```

```
23:counter_arr = M3;
24:counter_arr = V0;
25:counter_arr = L5;
26:counter_arr = V0;
27:counter_arr = L7;
28:counter_arr = V0;
29:counter_arr = M2;
30:counter_arr = V0;
31:counter_arr = L5;
32:counter_arr = V0;
33:counter_arr = L7;
34:counter_arr = V0;
35:counter_arr = M2;
36:counter_arr = V0;
37:counter_arr = L5;
38:counter_arr = V0;
39:counter_arr = L7;
40:counter_arr = V0;
41:counter_arr = M2;
42:counter_arr = V0;
43:counter_arr = L5;
44:counter_arr = V0;
45:counter_arr = L7;
46:counter_arr = V0;
47:counter_arr = M2;
48:counter_arr = V0;
49:counter_arr = L4;
50:counter_arr = V0;
51:counter_arr = L6;
52:counter_arr = V0;
53:counter_arr = M1;
54:counter_arr = V0;
55:counter_arr = L4;
56:counter_arr = V0;
57:counter_arr = L6;
58:counter_arr = V0;
59:counter_arr = M1;
60:counter_arr = V0;
61:counter_arr = L4;
62:counter_arr = V0;
63:counter_arr = L6;
64:counter_arr = V0;
65:counter_arr = M1;
66:counter_arr = V0;
```

```
                    67:counter_arr = L4;
                    68:counter_arr = V0;
                    69:counter_arr = L6;
                    70:counter_arr = V0;
                    71:counter_arr = M1;
                    72:counter_arr = V0;
                    73:counter_arr = L5;
                    74:counter_arr = V0;
                    75:counter_arr = L7;
                    76:counter_arr = V0;
                    77:counter_arr = M2;
                    78:counter_arr = V0;
                    79:counter_arr = L5;
                    80:counter_arr = V0;
                    81:counter_arr = L7;
                    82:counter_arr = V0;
                    83:counter_arr = M2;
                    84:counter_arr = V0;
                    85:counter_arr = L5;
                    86:counter_arr = V0;
                    87:counter_arr = L7;
                    88:counter_arr = V0;
                    89:counter_arr = M2;
                    90:counter_arr = V0;
                    91:counter_arr = L5;
                    92:counter_arr = V0;
                    93:counter_arr = L7;
                    94:counter_arr = V0;
                    95:counter_arr = M2;
                    96:counter_arr = V0;
                    default:counter_arr  = L1;
    endcase
```

Section 4: Simulation

4.1 Top model

The top is where we implement the main logic of the game, including battle solve, attributes change, what to display and so on. We have to admit that it is so complex that we didn't conduct simulation of the top module through the whole design. Instead, we debugged this part by downloading it into the board and operating the game.

4.2 VGA model

4.2.1 vga_display simulation

simulation code

```
initial begin
	data = 12'b111111111111;
end

always begin
	clk = 1;#20;
	clk = 0;#20;
end

always begin
	vidon = 0;#100;
	vidon = 1;#300;
end
```

simuation waveform

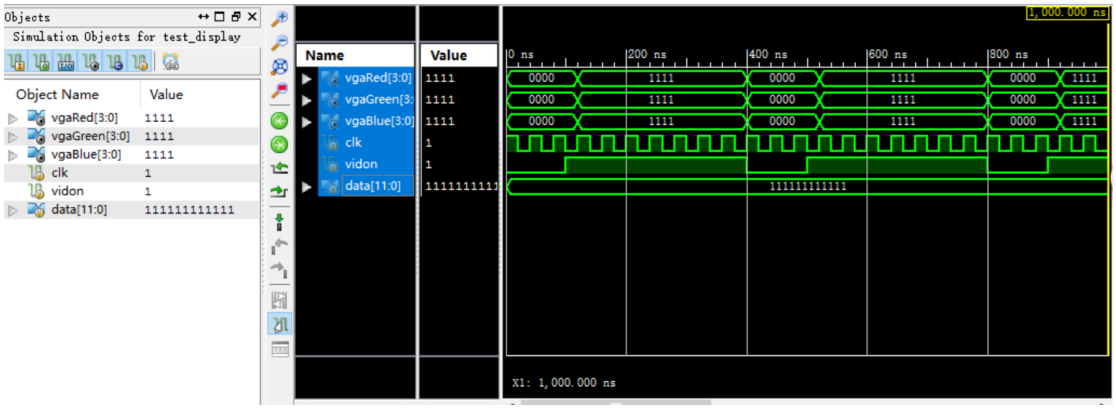


Figure 17 Simulation result of vga_display

we can see from the waveform that when vidon = 1, the value in data is copied to RGB, so it is

workable.

4.2.2 vga_sync simulation

simulation code

```
always begin
        clk = 0;#1;
        clk = 1;#1;
end
```

simuation waveform

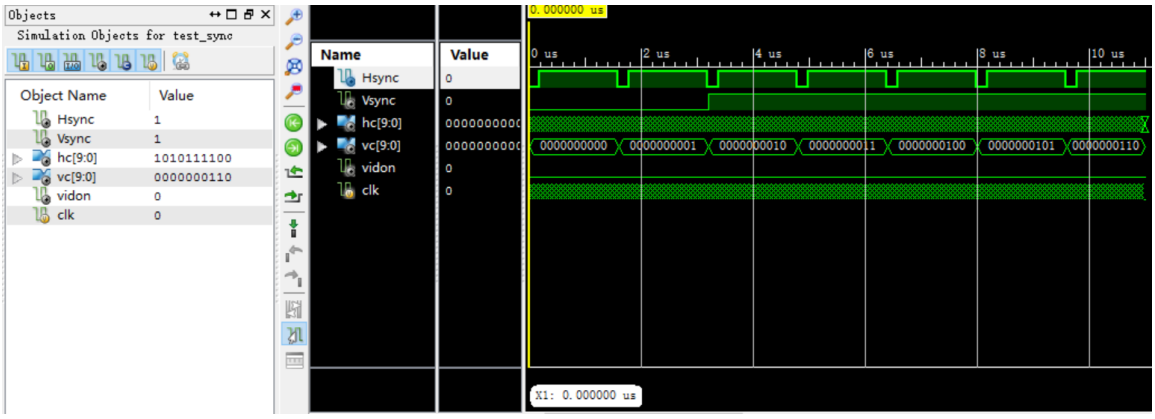


Figure 18 Simulation of vga_sync (1)

We can see in the waveform that the Hsync is changing from 1 to 0, the same as the timing diagram in chapter3. The Vsync changes to 1 at the beginning and will not get to 0 until the whole monitor has been covered. The hc changes much faster than the others (synchronized with clk). And we can see each time vc is added by 1. Here we cannot see the vidon to be 1, because all of this are in boarder. If we make the time longer, we can see the vidon = 1 after vc > 31 like the diagram shown below.

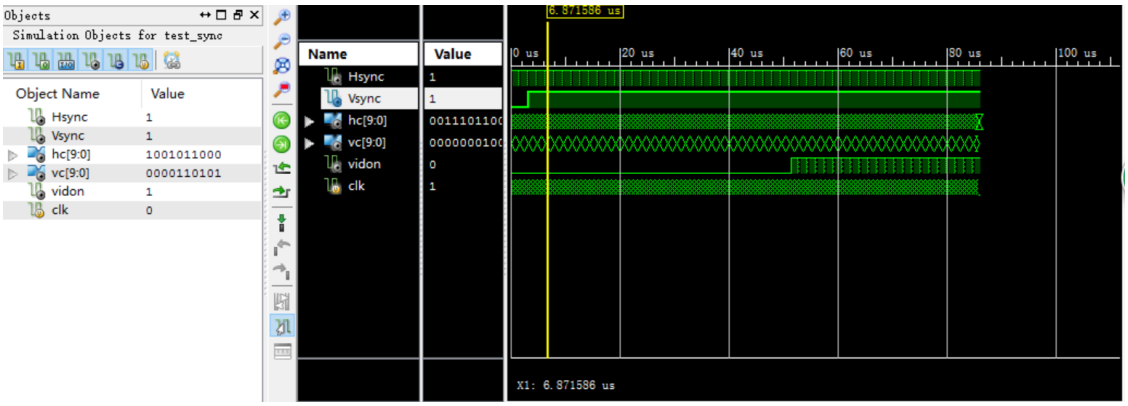


Figure 19 Simulation of vga_sync (2)

4.3 Buzzer model

The most important part of the buzzer module is pwm_generator, so we just give the simulation of this part to show the code is correct.

Here we have the clk is 500MHz and set counter_arr as 99 and counter_ccr as 50 to generate a PWM wave with 5MHz and 50% duty cycle. The simulation result is shown below.

```
initial forever begin
        #1 Clk50M = 0;
        #1 Clk50M = 1;
end
initial begin
        Rst_n = 0;
        cnt_en = 0;
        counter_arr = 0;
        counter_ccr = 0;
        #50 Rst_n = 1;
        counter_arr = 99;
        counter_ccr = 50;
        #100 cnt_en = 1;

end
```

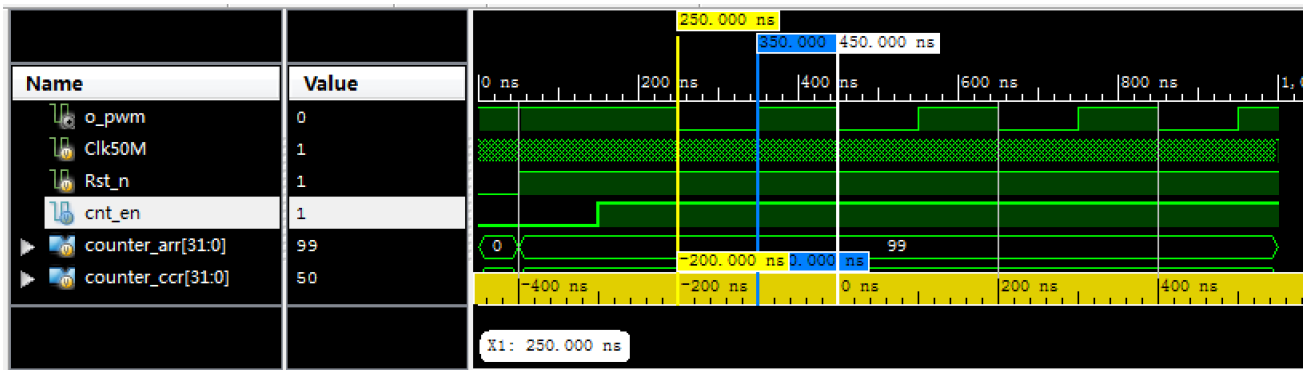


Figure 20 Simulation of Buzzer

Here when Rst_n is 0, the counter of frequency divider resets to 0, and when cnt_en is 0, the circuit won't work. When Rst_n and cnt_en are all 1, the counter begins to count, and the circuit can generate PWM waves as the above picture shows. Here we set counter_arr as 99 so that we can generate a wave with 5MHz frequency (although we cannot hear a sound with so large frequency) and set counter_ccr as 50 so that we can make the duty cycle be 50%.
The o_pwm is the output and we can see that its time period is 200ns which means its frequency is 5MHz and the time periods when it is 1 and 0 in one cycle are all 100ns which means its duty cycle is 50%. As a result, we can come into a conclusion that the module of pwm_generator is correct.

Section 5: Debug and analysis

Actually, the overall design idea is to design by block, debug by block and integrate finally.

5.1 Main logic

When designing the main logic code, we encountered many problems, but due to the very slow integrated speed and achieved speed of Verilog, we conducted debugging the logic code by checking static code on the whole.

Through checking the context and deducting the logic of state, we can correct the obvious logic errors. Some of the problems on the movement were mostly due to the input errors of the constant, while the other problems were mostly due to the timing problems. In order to solve the problem of it, we used to increase the free cycle. Actually it will reduce the efficiency of parallelism, but at Verilog, this does not cause a significant impact on our users' experience. In addition, for some logic errors which were difficult to solve, the simulation of behavior description level had been insufficient to find problems, so we adopted the timing simulation after synthesis, in which case we could also find potential timing problems when there was no hardware condition to debug.

Through implementing after simulation, we found that the IP core had certain timing delay when the enable signal was 1, so we increased the idle cycles, and finally succeeded in solving the problem of displacement. In addition, there were some problems that map boundaries were not inconsistent with the data in the memory, so we outputted the data to the seven-segment digital for debugging, and later found that the display part of the combinational circuit was out of the question.

5.2 VGA model

Through learning the implement of the principle of VGA, we found that the most important operation is the signal synchronization. Having adopted the idea of modular programming, we divide the main implement of VGA module into two parts, vga_display and vga_sync. First of all, we discuss the vga_display module.

5.2.1 vga_display module

The vga_display module is relatively simple to implement. Our goal is to determine whether can it transmit the corresponding RGB value to the display by checking whether can it display the signal. So in the actual programming did not encounter obvious problems. The test of this module is showed in the simulation, when the vidon signal is 1 the value of all the RGB is copied, vidon is 0, the value of all the RGB set to 0.

5.2.2 vga_sync module

The vga_sync module compared to vga_display module is more complicated.
1. First of all, the problem we encountered is the determination of various parameters. Referring to the VGA detailed description of the scan line scan parameters given by teacher, we found around there is a certain black border and a difference of the thickness. So we fine-tune the parameters to make sure the black boarders are same in thickness.
2. The timing problem of regulation. The frequency of the scanned pixel is 25 MHz. But for the line synchronization signal, the column synchronization signal and vidon signal we use asynchronous real-time updates. Initially we used the clock signal so that cause the serious timing question. Because the synchronization signal cannot be controlled by the clock.
3. The test samples are given in the simulation. For the test of this module we only need to provide a 25MHz frequency signal to see the line scan line scan signal continues to move.

5.2.3 The simulation of display

Before we are ready to display the map and sidebar, we first want to show some of the simpler images to test.

1. Black and white checkerboard display
We hope that through the black and white checkerboard grid to display the map grid to do some simulation, then each grid is composed of monochrome (black and white) form, is relatively simple.

```
initial begin
      R2[0] =   12'b000000000000;
      R2[1] =   12'b111111111111;
end

always @ (posedge clk25) begin
      if(vidon == 1)
            if((hc-143)/16%2 + (vc-34)/16%2 == 1 )
                  data = R2[0];
            else
                  data = R2[1];
end
```

2. WIN icon display
We want to be able to display some of the larger tip of the text interface, such as the end of the game WIN.

```
reg [11:0] frame[0:20'h60000];
 always @ (posedge clk25) begin
      if(vidon == 1)
            case((hc-143)/16 + (vc-34)/16*40)
                  430,470,510,550,590,630,670,710,
```

```
                750,790,435,475,515,555,595,635,
                675,715,755,795,420,460,500,540,
                580,620,660,700,740,780,431,472,
                513,554,360,400,401,441,442,482,
                483,523,524,564,530,570,571,611,
                612,652,653,693,694,734,726,686,
                687,647,648,608,609,569,615,575,
                576,536,537,497,498,458,459,419:data = R2[0];
                default: data = R2[1];
        endcase
end
```

3. The picture display

The image contains a large data stream, and we want to simulate the process by displaying the image.

```
reg [11:0] frame[0:20'h60000];
    initial begin
        $readmemh("lovelive.mif", frame);
    end
always @ (posedge clk25) begin
    if(vidon == 1)
        data = frame[(hc-145) + (vc-32)*640];
end
```

5.3 Keyboard model

We tested our ps2 code by displaying the key code on the 7-segment display decoder. As we expected, it works very well. When a key was pressed, its key code will be displayed on the 7-segment display decoder in hexadecimal form until another key was pressed. But when we use the ps2 module directly in *top.v*, we found that the character will not move until a different key is pressed, which means the character moves when we first press ↑, but it can't move until we press another key rather than ↑. So we introduced a sequential circuit to manage ps2 inputs and finally solve this problem.

Keyboard is very useful as an input equipment. It's much more user-friendly than switches. Before we mix the ps2 module with VGA, we use the switches as input. Every time we tested VGA, we had to continuously switch the switches on and off. It's so boring and tiring! But after applying ps2 module to *top.v*, it's much easier for us to control the movement of the character.

5.4 Buzzer model

To play music with a buzzer, the first thing we need to do is that we must calibrate the frequency of tones the buzzer lets out with the real frequency. So, after the buzzer can let out sound, we first designed a program to make it let out do, re, mi, fa, so, la, si every half of second. Actually, the

frequency was accurate on the whole but when we played music using buzzers on different boards, the actual performance was different. The reason is that different buzzers have difference in materials, structure or other details which we cannot control.
The code which implements playing notes do, re, …, si as a loop every half of second.

```
//500ms 延时计数器计数
always@(posedge Clk50M or negedge Rst_n)
if(!Rst_n)
        delay_cnt <= 25'd0;
else if(delay_cnt == 0)
        delay_cnt <= 25'd24999999;
else
        delay_cnt <= delay_cnt - 1'b1;
```

After we had calibrated the tones, to play music we had to control the beats which is shown as a counter. The greater the counter is, the longer the beat period is. We had tried 2Hz, 5Hz, 10Hz, 20Hz and finally according to the requirement the music need we chose 10Hz.
As an example, the implement code of 5Hz is as follows.

```
always @(posedge Clk50M,negedge Rst_n)
        begin
                if(!Rst_n)
                        cnt <= 25'd0;
                else
                begin
                        cnt <= cnt + 1'b1;
                        if(cnt==25'd9999999)
                                cnt <= 25'd0;
                end
        end
```

Section 6: Conclusion

Firstly, we give the final conclusion.
We implemented a game called Magic Tower based on Verilog HDL which includes 85% function of the original and we made use of a keyboard, a monitor and a buzzer as accessories besides the development board; however, limited by time and hardware condition, there are still room for improvement in our design.

1. The memory is not enough
Limited by the condition the development board can provide, we can just finish 80% of the complete Magic Tower game. The prime limit is the memory. At the beginning, we also tended to design the beginning interface and the story plot of Magic Tower. But we had occupied almost all of the memory the board can supply when we finished the main logic of the game and the fundamental interface, so we had no chance to show the complete Magic Tower. However, we have designed the beginning interface and we really want to show it here.

Figure 21 Beginning Interface

2. The design is not modular enough
Another aspect we can improve is the code. Due to the lack of time and experience, the code has somewhere that can be optimized. We have tried our best to do the modular design which means in other situation the single module, like VGA or the main logic, can be used after only a little modification. However, the end project we give still has shortcomings in this regard that some of the codes have been integrated together. So as we talk about in section 3.1, the MVVM design model we make use of still has room for improvement.

Section 7: Team member and contribution

Zhang Chengyi: Design the main logic of the game and provide technical support.
Lu Jikai: Design VGA model and the display of the game.
Wen Zihao: Design the PS/2 model and write the report.
Wang Daxin: Design Buzzer model, produce the model materials and write the report.

Contribution ratio
Zhang Chengyi: 25%
Lu Jikai: 25%
Wen Zihao: 25%
Wang Daxin: 25%

References

1. Magic Tower Game: *http://www.4399.com/flash/1749.htm#search1*
2. The PNG martials of Magic Tower from the Internet
3. Logic and Computer Design Fundamentals, Fourth Edition, M. Morris Mano and Charles R. Kime
4. 夏闻宇. Verilog 数字系统设计教程. 北京航空航天大学出版社, 2008
5. FPGA 无源蜂鸣器驱动设计:
*http://wenku.baidu.com/link?url=enzxE3EWO_Eo5nrQnHouBteOVx0ATWfxTuvQ3xWrutycDv4JyUtO8BuD_41gWjfnmX0ES2BljeTwp3cjOCWEVdYBJS6TSQ5x5geUPpGmx1_*