WormCraft

CSEE 4840 Embedded System Design

Tianyi Zhang	tz2210
Ning Li	nl2443
Yuxuan Zhang	yz2580
Ziwei Zhang	zz2282

Overview 4
System architecture 6
VGA display module 7
Mif and ROM 11
Decoder design 11
VGA Display 12
The SW/HW interface 13
Software 14
User interface 14
The overall game control 14
The map and craters 14
Element blocked 15
Player move 15
Bomb 16
Game over 16
Audio 18
Input Control: Wii Remote Controller 20
Basic Implementation 20
Experience and Issues 23
Timing Violations when implementing the crater Effect 23
Large Delay of Wiimote Update 23
Priority of Image Display 24
Lesson Learned 25
Responsibilities: 27
Ning Li: 27
Tianyi Zhang: 27
Ziwei Zhang: 27
Yuxuan Zhang: 28
C Code 29
main.c 29

object.h 38
connect.h 41
map.h 44
varibale.h 47
bomb.h 50
move.h 56
reset.h 58
message.h 60
display.h 62
System Verilog code 64
VGA_LED_Emulator 64
VGA_LED.sv 83
audio_effects.v 94

Overview

In this project, we intend to implement an “artillery strategy” video game like Worms®. The player needs to throw bomb toward each other within a limited time in order to hurt the enemy controlled by another player. The player can move forward, backward and jump to pick the appropriate position to toss the bomb. The bomb will not only hurt the other player but destroy the ground as well. So to bombard the rock to create a favorable geography is also a highlight of this game. If one of the player runs out of life, then the game ends with the other player winning. The player can select three types of bombs which have different effects on the player and the ground. And various landforms are also performed so that the player will have more flexibility to choose the right strategy to pinpoint the best explosion position.

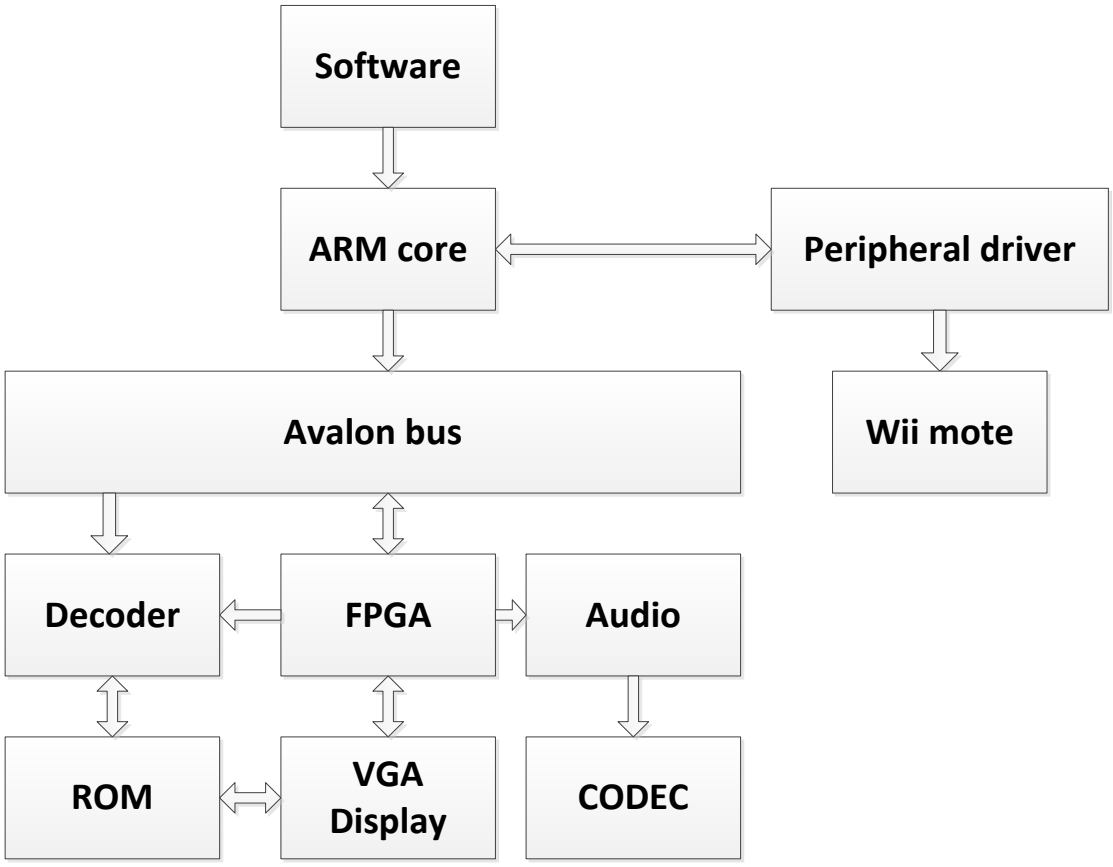
There are several key challenges of this project that we need to handle carefully. First, since there are many types of elements in this game, such as player, landform, HP/Time/Strength bar, multiple landforms and pointer, the display of those elements will collide.

Second is the algorithm issue. When the player walking or jumping, it may hit the rock or explosion remnants and thus be blocked. And the player may also fall from high platforms so that it has to move downward by gravity. Also when the grenade hits the wall, it will be bounced back following the momentum theory. And after the explosion, to correctly display the crater requires much attention. So the implantation of these algorithms is the hardest part of this project.

Third is the operation of Wii Remote Controller, or in short, wiimote, which stands far away from lab2 and needs to be completely rewritten for this project. Lots of layer problems between hardware and operating systems such as bluetooth driver and wiimote driver have to be solved.

System architecture

Our system structure is shown below:



Our design includes several key subparts. The VGA display module contains the decoder, ROM, VGA display. Decoder receives control signals from software and display constraints. It delivers the address of data saved in ROM and the 24 bit VGA color value. VGA display implements all the component displays which follows the design constraints and get color value from ROM. The Audio module contains audio controller, which provides different sound effects triggered by displayed components, and CODEC part to play the sound. Wii mote connects to the core via Bluetooth and uses peripheral driver to provide service for communication.

VGA display module

Image category

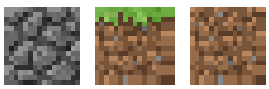
In order to get the pixel style display and show components on the screen correctly, we should do some preparation for all images. Firstly of all, put all the figures in the single color background and sizing the images follows the game logic design. With this parameter, we can calculate the coordinates of every component and define the display range.

In our design, for the best display of the game effect and save memory space as much as we can. We used several types of elements to be displayed with diverse definitions.

1. Ground block
2. Figures
3. Pointer
4. Bombs
5. Explosion
6. Life/Timer/Strength
7. GameOver show
8. Background

* Crater

1. Ground Block



Ground blocks can be used to form the ground in the game. There are three kinds of ground block: grass, soil and stone. Each of them is sized of 16x16

pixels. When a bomb is exploded in the bound range, the grass and soil blocks will be destroyed, leaving a crater there. The stone blocks cannot be destroyed by the bomb.

2. Figures

The two figures can be controlled by player to move, jump and throw bombs. Each figure is sized of 44x44 pixels. To meet the motion design, each figure has 5 different pictures to show all the action. The figure will get injured when an explosion happened around. Since it's a turn-based system game, when times out or one figure throw the bomb, the current player will hold and the other one acts.

3. Pointer

This 13x13 pixels component is shown with the figure who has action right in the turn. The left/right direction of the pointer is same with the figure orientation. It can also move up/down with the coordinate data returned by G-Sensor of the Wii mote controller.

4. Bombs

Bombs are weapon throw by the figures. We set three kinds of bombs. All of them share the same size 16x16 pixels. The grenade will explode whenever it

hit the ground block or player. The timer will explode within a set time after being throws out. The last one is missle which will explode only it hit the ground blocks. The trajectory of bomb is parabola. The angle of throwing follows the pointer direction and the distance is based on the strength value.

5. Explosion

When an explosion happened, there shows explosion effects. Explosions will last a few microseconds shown the explosion effect images from explosion1 to explosion9. Each picture is 64x64 pixels. Explosions will destroy ground blocks and hurt players.

6. Life/Timer/Strength

We put each side three bars to show the life value, left time for play, strength value. The size of them is based on the length. When the player gets hurt, the life value will decrease. The left life value is shown in the bad with the change of color from green to red. The length of timer bar decreased with time elapsing. And the strength bar displays the power of throwing bomb with gradient color.

7. Gameover

It's a K.O. picture shown in the screen when one figure when the game. This scene is 139x340 pixels which would occupy much memory space with full 24bit color. In order to save the memory, for it's a dual color pic so we convert it into 1bit width mif only choose to show the words or background. It could to great extent save memory space.

8. Background Scene

The background image shows us different scenes where battles happen. It's a 128x512 pixels file so we reused it to save the on chip memory space. This scene is put in the lowest layer below other sprites. So it cannot be destroyed

as by the explosion of bombs like ground blocks.

9. **Crater**

This one is quite special. Since we should make a circle as the crater caused by bomb explosion, the time cost of calculation, which will greatly affect game performance, is significant for design. In order to save the time cost and prevent timing violations, we get the 64x64 pixels circle by defining a two-dimensional array, assign value to each bit in the component. The implementation of crater is to invalid block display and show the background.

Table for all the components

Image category	File name	# of images	Single image size(pixels)	Total size (Byte)
Stone	stone	1	256	768
Grass	grass	1	256	768
Soil	soil	1	256	768
Figure No.1	stand1	6	1936	17424
Figure No.2	stand2	6	1936	17424
Pointer	pointer	1	169	768
Head1 portrait	head_p1	1	2090	12288
Head2 portrait	head_p2	1	2090	12288
Explosion	explosion	9	4096	110592
Gameover	ko	1	47260	8192
Background	background	1	65536	196608
Total		29	125881	377888

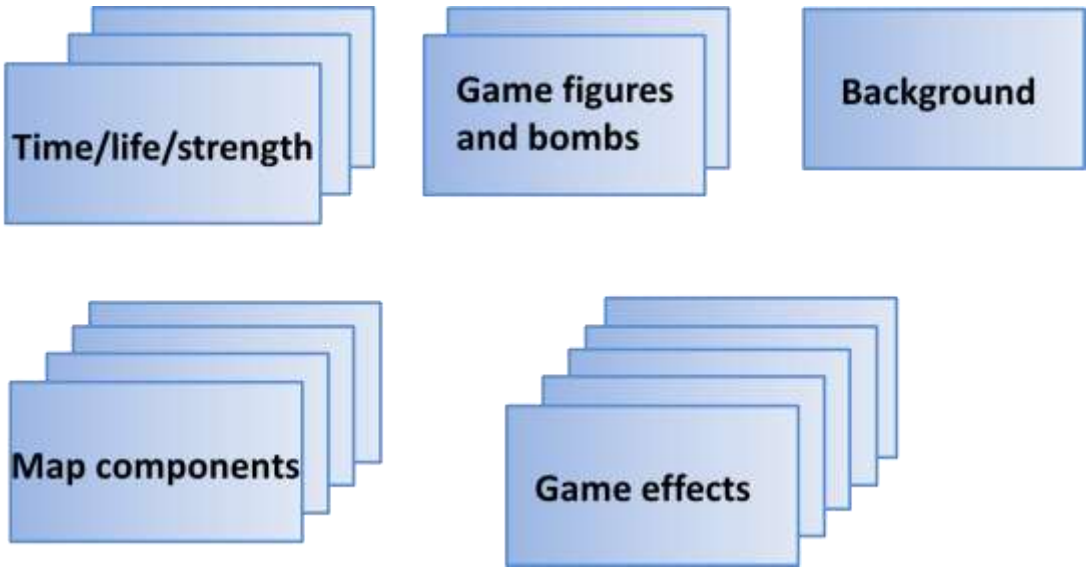
Mif and ROM

Then we should use MATLAB to convert the image to mif file. It can specify the initial data of the memory block in the chip. This file has the same parameters with original pictures such as number of rows and columns and the 24 bit RGB color value. Each line of the mif holds the actual binary data saved in the memory. It consists of one line of data per memory location and the line number indicates the address in memory. We use Mega-Wizard Plugin to add new ROM initialized by specified mif file. It will generate a Verilog file explains the parameters. We design decoder to provide memory address as input to read data from the ROM.

Decoder design

After generating all the mif files for components, we should design various decoders to provide 24bit VGA RGB values of the specified component. There are several types of inputs. Firstly, we utilized the VGA screen scanning at 50MHZ clock to get the hcount value and vcount value at one time, which can be used as inputs to set the starting pixel of the picture. We can also define the deviant value from the original coordinates to set the location of specified component in the screen. Moreover, the vertical and horizontal coordinates sent from Software can be used as input. It can calculate the specified coordinates or set as conditional judgments. Such as make picture display choice or choose the direction of component. The output data of the decoder is 24-bit RGB pixel

Level of sprites



VGA Display

In order to achieve the portability of the program and make it easy to debug, we make diverse subprograms to implement all the logic design. So it's easy to add new module into the project design. With the help of Decoder, the main function, VGA display module saves a lot of work. What it needs to do is to invoke sub-functions and get the RGB information for specified picture from decoder, then display every pixel on the screen. For different components, we define coordinates constraints to various display range. Since we have several levels of display layers with priority, in the VGA emulator function, we write conditional judgment to decide which sprites can be shown in the screen. The figure to show the level of sprites is shown above.

The SW/HW interface

The communication between software and hardware is the key for game logic. So we set sixteen 16-bit hex arrays to transfer control signals from the software part. The table of the hex and control data is shown below

	15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Hex0	Number of crater						Vertical coordinates of bomb									
Hex1	Explosion choice						Horizontal coordinates of bomb									
Hex2	face	Fig1 pic show					Vertical coordinates of figure No.1									
Hex3	Health display of figure1						Horizontal coordinates of figure No.1									
Hex4	face	Fig1 pic show					Vertical coordinates of figure No.2									
Hex5	Health display of figure2						Horizontal coordinates of figure No.2									
Hex6		Fig	EN	up/ down	Vertical coordinates of pointer						Horizontal coordinates of pointer					
Hex7	K.O.	Time counter for 30sec					Strength of throwing									

Software

User interface

After download the program and running it, the game will start. A big map will show on the screen. We use kinds of blocks grass, soil to form mountain, plain and stairs, which can be used to jump to the higher place. Two players stand on left side and right side of the ground. The top of the map is information bar, which shows the current HP, Strength and time left of each of the two players.

In our design, software part needs to handle the following situations.

The overall game control

All of the control signals need by FPGA are generate from software, including the sound control signal: indicate when and which sound to play, crater counter signal: indicate the number of the crater been created, player and bomb picture select signal: indicate which pictures of the bomb type (grenade, missile, timebomb), player (stand, run, jump, throw, fail) and explosion effect (#1~#9) we wanted to display on the screen. The location of all elements are also generate from software, as shown in the table left.

The map and craters

We use a two dimensional array (480*360) to record whether the ground is valid or not at a certain pixel location, ‘2’ for *stone* block valid, ‘1’ for *grass* and

soil block valid and “0” for there is no ground here or previous valid ground pixel has been invalid by an explosion. To create a crater, we just set the array value near the explosion location to “0”, note the stone block can’t be destroyed by explosion, which means that only ‘1’ can be set to ‘0’.

Functions related:

map_init()

press_bomb_post()

Element blocked

When an element (player, bomb) move toward and touch a valid ground, we call that this element is blocked on this direction. To indicate one element has been blocked or not, we check all of the four edges of the element, if any number pixel of ground is valid in that edge’s location, we can announce that the element has been blocked on this direction. In our game, players can’t move forward to the blocked direction, and the bomb will reflect when it been blocked during a movement.

Functions related:

if_bomb_block()

if_player_block()

Player move

We use three keys to control player’s movement, left, right and jump. Players can move only if that direction is not blocked. Also, if there is no blocked at bottom, player will fall until hit the ground.

Functions related:

if_player_block()

player_action()

Bomb

There are four stage of bomb throwing: change type, change angle, change strength, and toss. We have three types of bombs: grenade, missile and timebomb. Grenade can reflect several times before explode, missile will explode as soon as it hit the ground, timebomb can't move after been put on the ground it will explode a few seconds later. Note that each player has ten bombs at most.

Functions related:

bomb_change()

press_bomb_pre()

press_bomb_post()

if_bomb_block()

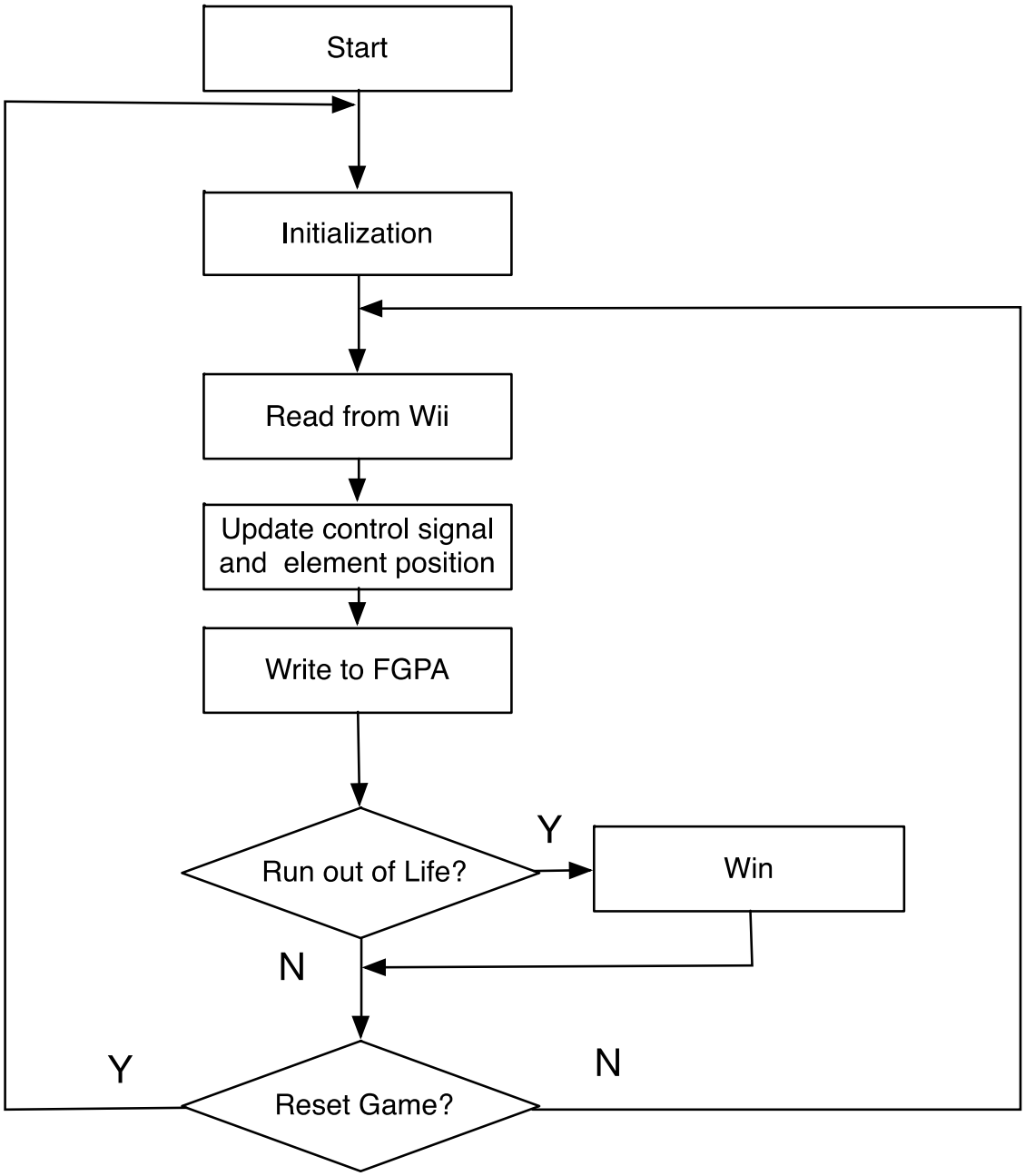
bomb_bounce()

Game over

When a bomb explode near a player, it will decrease the HP of this player. HP lost is determined by the distance between bomb and players. If one player's HP is under 0, the other one will win.

Functions related:

game_over()


Start
Initialization
Read from Wii
Update control signal and element position
Write to FGPA
Run out of Life?
Y
Win
N
Y
Reset Game?
N

Audio

Our project will play sound effects for strength increasing, jumping, tossing bomb and explosion. The source audios are extracted from an Action RPG Game in 1993 called Gunstar Heroes.

We use audio codec on SoCKit board to play the sound effects stored in the FPGA ROM. These enable signals are indirectly controlled by the signal output from the software, directly passed from LED_Emulator driver. In our design, VGA display module (lab3) and Audio_Top mode are two separate hardware in the top architecture. Since the Avalon bus is responsible for the data transmission from the software parts to hardware parts and it is already implemented in the driver of LED_Emulator, the signals datapath from variables defined as “hex” in the driver have to be transmitted to the top level of the whole design as an output and then input into the audio_top level. So the audio part is “activated” in this way.

Inside audio_top, there are four parts in our design, including audio_codec module, i2c_av_config module, audio_effect module and a PLL for sample rate clock generation. The aducio_codec defines parameters of SSM2603 on board, specifically the master clock MCLK, bit clock BCLK, RECLRC and the left clock and right clock for two stereo channels RECLRC and PBLRC. These clock signals are related with the clock sample rate respectively. The sample rate clock is 44.1kHz which generated by PLL using Mega wizard in Quartus. I2c_av_config drives the i2c_controller inside it. I2c_controller specifically define the communication path between FPGA as a master and audio Codec as a slave based on I2C protocol. We modify the transmitted signals as 7-bit address, 1-bit read/write enable and 8 bit data. The driver specify the several requirements and we basically increase the left and right sound volume into

6dB.

In audio_effect module, we specify our sounds. All sounds are stored in ROM and used in this module. To simply the design, we use “hex” signals indirectly controlled by software “write_data”. The signals are linked coherently to the particular sprite displays. Specifically, the 15 to 12 bits of “hex1” indicates the index of explosion figures. These bits are also used to enable the sounds of explosions. Similarly, we used the same idea to enable other sounds. Sound playing at the same time cannot overlap. So the explosion is on the highest priority, then the tossing bomb, then jumping, then strength increment. In order to better control sequences our sounds and reset the address signals input into the sounds in ROM, we use a FSM for controls. Basically, all of the sounds effects last for the time when index increases from 0 to 65535 if flag is zero (16-bit address of sound ROM) and the controller asserts a flag to 0 if it accumulates to 65535 or expires much soon because of the short period of enable signals from software. When a new sound is issued, the flag bit will reset to 1 and begin to play the sound. With this controller, our sounds are played smoothly.

Input Control: Wii Remote Controller

Conventionally, the game HCI is based on joystick or gamepad, whose transmission protocol is the same as keyboard. However, for this game, the strength and launching angle has to be set every time before tossing a bomb. A joystick with gravity and acceleration sensor will improve the user experience a lot if we don't have to press any button to change the launching angle. So a wiimote is used as user input in our video game for a better graphic user interface.

Basic Implementation

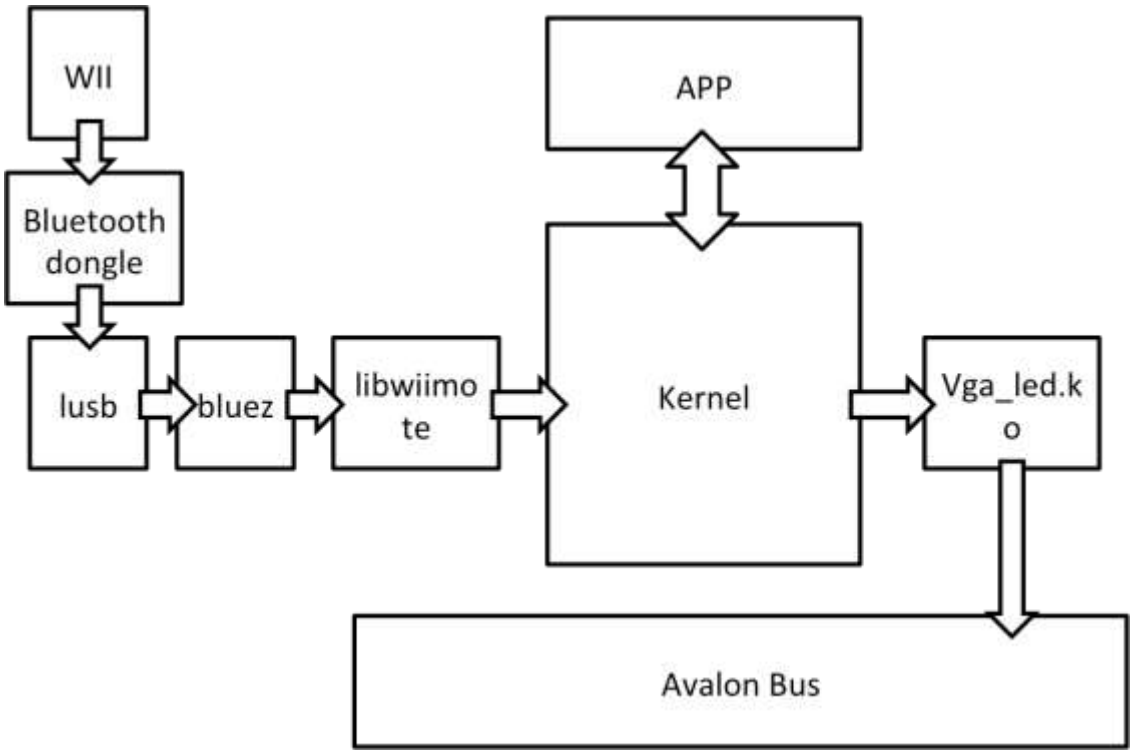
Unlike keyboard or other joysticks, wiimote is connected to the HPS core wirelessly through bluetooth RFCOMM protocol. So special consideration toward this new method of connection should be taken.

There are three protocol stacks between wiimote and application layer. First is between wiimote and bluetooth dongle. Second is between the bluetooth dongle and the HPS. Third is the driver to decode packet of wiimote to our desired data format.

Once the software is launched, the program will call the wiimote driver "libwiimote". The wiimote driver will then call the bluetooth protocol stack "BlueZ" to start searching the bluetooth device around through the bluetooth dongle with the given MAC address. If such a bluetooth device is found, then the pairing between wiimote and SoCKit board (host) will occur. If the pairing succeeds, the host begins to receive data packet from the wiimote. In our case,

the process “receiving data packets” is accomplished by calling wiimote.update(&wi) function.

The data packets include the state of all buttons (i,e four direction buttons, button A, B, +, –, and home), the state of gravity sensor (the gravitational force on the sensor in x, y, z, direction), and the state of acceleration sensor (the newton force F=ma on the sensor in x, y, z, direction). The wiimote data transferred and processing is done in the software. The calculated acceleration will be calibrated to fit into our design and giving the player the corresponding action.



The launching angle is realized by tilting the wiimote. The exact angle of tilting in y direction is read from the gravity sensor. The minimum and maximum value for that tilting angle is 0 and 180. So when y = 180 the launching angle is pointing upward, 90 for horizontal and 0 for downward. That launching angle will be changed when pressing the B button. If B button

is pressed, the pointer indicating the launching angle will change its position along a circle centered at the player according to the tilting angle. Note the maximum and minimum boundary in our design for that tilting angle is slightly larger than 0 and slightly smaller than 180 to avoid potential errors in C programs <math.h> library. The current value for the tilting angle will be updated every 8ms to ensure the correctness of game program and display. And the wiimote.update(&wi) function is called as often as possible in our design,

The tossing flag is realized by wielding the wiimote at a very high speed. Since the acceleration sensor in the wiimote is too sensitive, it will hit the maximum value 244 very easily. To ensure correctness, all three coordinates of acceleration sensor will be used. If any of them, in x, y, z direction hit the maximum value, tossing flag will be held at positive.

The other buttons, four direction buttons, button A, B, +, –, and home will also be updated as often as possible. The strength of tossing bomb will be increasing as long as button A is pressed. And the launching angle will be adapted to your tilting angle if button B is pressed. The type of bomb will be changed if button + or – is pressed. And the game will be ended and re-initialized when button home is pressed.

Experience and Issues

During the entire design, we encountered all kinds of obstacles which once made us distressed and desperate. Finally they are all solved.
Here are some examples from which we learned the key design principles.

Timing Violations when implementing the crater Effect

When we tried to implement the crater effect, at first we used a large combinational logic to calculate the distance from the explosion center to all data points on the screen to make sure that all points within the crater are within the explosion radius, whose calculation time is much larger than one clock cycle. Then we tried to roll up the pipeline to calculate that in multiple clock cycles, which still didn't solve the problem. After thinking about the solution for over a week, we exchange space for calculation time, and put a predefined circle into the memory. So every time the explosion occurs, the only calculation will be determining if the points on the screen are within the 64x64 square centering at the explosion center. And the timing violation and noise never happened after then.

Large Delay of Wiimote Update

When we tried to use the wiimote to control the player in the game, the action of the player is about 0.5s later than the moment I pressed the corresponding button. The reason for that is the condition of the infinite while loop. For an

infinite loop I put the condition of “while the wiimote is on”, which means before every loop begins, the system has to check whether the wiimote is turned on, which costs extra time for re-pairing. So no time-consuming conditions should be defined within the infinite loop.

Priority of Image Display

When we were designing the sprite controller, we found that at some spots, where should display the rock, actually displayed the background. The reason is we didn’t put the background in low priority of display. Even if we set the boundary for the blocks, when we put player action inside, the display will change anyway. So giving the picture tile the correct priority is essential.

Lesson Learned

After one semester's study and work on the Alter Cyclone V SoCKit Board, we are familiar with the architecture of System-On-Chip with integrated FPGA and HPS core. And our coding skill in SystemVerilog and C is greatly enhanced. Now we have got the proficiency in using FPGA design suite and CAD tools in Altera Quartus.

Top-down Design: To improve the Verilog and C code readability, the abstraction method should always be used. Rolling up the design will greatly shrink the length of main function, and the abstracted functions are much easier to read than simply put long comments within the code.

Version Control: For any updated design, never remove the previous versions. Any modification on the previous code may modify even destroy the original function of the code. Keeping the original version will help recover the needed functions. Also keeping comparing the two versions of code will help improving the algorithm.

Make Plans Ahead: Always think about as many details as possible before the project starts. Before doing this project, we have already decided to use the pixel sprites from MineCraft, figures from Gunstar Hero. So the picture clips are prepared very early. Also for the controller we planned to use Wii Remote Controller right after the team is formed. So the testing of driver, which costs tremendous time can be accomplished before MileStone 2.

Always Do the Timing Constraint Analyzer: Since the logic units are used extensively in this project, and for a lot of algorithms the combinational logic

is unavoidable, the timing constraint becomes an important concern. So before compiling the project, always analyze the timing constraint so make sure that all computations can be finished within one clock cycle.

Using Break Points: To debug the System Verilog code and C code, the best way is to set a interrupt, or a break point, and return all the variable values. In C, we used gdb to debug. In System Verilog, ModelSim is strong enough to check all signals' value at a specific time.

Work Independently and cooperate wisely: When someone finished his part, others should not try to modify his code to accomplish his function before he completely understood what he was doing. The interaction between team members are essential. By cooperating wisely, a lot of repetitive work can be saved, and the project will also be implemented in pipeline to improve efficiency.

Responsibilities:

Ning Li:

Writing the software part of project.

Wiimote Integration to software.

Clock domain organization.

Debugging in software and VGA.

Tianyi Zhang:

Co-write the software part of project and code improvement.

Recompiling linux kernel, setting up bluetooth and wiimote driver.

Coordinating the hardware-software interface.

Ziwei Zhang:

High level hardware design.

Sprite placement and map design.

Group organization.

System interconnection between components.

Yuxuan Zhang:

All audio components.

Picture clip of figures.

Implementation of several functions in hardware.

Building the Memory Initialization modules.

C Code

main.c

```
#include <stdlib.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <math.h>
#include <sys/socket.h>
#include <stdio.h>
#include "vga_led.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include "wiimote_api.h"
#include "wiimote.h"
#include "connect.h"
#include "map.h"
#include "move.h"
#include "variable.h"
#include "reset.h"
void print_segment_info();                          // Read and print the segment
values
void write_segments(const unsigned char segs[16]);    // Write the contents of the
array to the display
```

```
int main(){

   Player p[2];
   Player p[0] = makePlayer(53,346,RIGHT, 200);          // Initialize Player 1
   Player p[1] = makePlayer(597,345,RIGHT, 200);         // Initialize Player 2
   Bomb b = makeBomb(0,0,GRENADE);
   Pointer ptr = makePointer(0);

   while (1){
      connect_wii();
      connect_vga();
      map_init();
      crater_rst();
      reset_game();

      while (1){

         update_wii;    // synchronize with wiimote
         press_wii();
         resetBlock();
         for(n=1;n<15;n++) {
            bombUpBlock    += map[b.y-7][b.x + n-7];
            bombDownBlock += map[b.y-7+14][b.x+n-7];
         }
         for(n=1;n<14;n++) {
            g_left_blocked    += block[y_grenade-7+n][x_grenade-7];
            g_right_blocked += block[y_grenade-7+n][x_grenade+14];
         }
```

```
if (direction == LEFT){
    for(n=0;n<44;n++){
        leftBlock    += block[p[in].y-21+n][p[in].x-21+0];
        rightBlock += block[p[in].y-21+n][p[in].x-21+35];
    }
    for (n=0; n< 35; n++){
        upBlock    += block[p[in].y-21-1][p[in].x-21+n];
        downBlock += block[p[in].y-21+44][p[in].x-21+n];
    }
}

if (direction == RIGHT){
    for(n=0;n<44;n++) {
        leftBlock    += block[p[in].y-21+n][p[in].x-21+9];
        rightBlock += block[p[in].y-21+n][p[in].x-21+44];
    }
    for (n=9; n<44; n++){
        upBlock    += block[p[in].y-21-1][p[in].x-21+n];
        downBlock += block[p[in].y-21+44][p[in].x-21+n];
    }

}

stand(&p[out]);

if (pressRight){
    if (!rightBlock && p[in].x < 597)
        moveRight(&p[in]);
    else
        moveRightBlock(&p[in]);
```

```
}

else if (pressLeft){
    if (!leftBlock && p[in] > 0)
            moveLeft(&p[in]);
    else
            moveLeftBlock(&p[in]);
    stand(&p[out]);
}
else {
    stand(&p[in]);
}

//************** jump ************************
if (pressJump & down_blocked!=0)
    jump = 1;
if (jump){
    if (upBlock)
            jump = 0;
    else if (jump < 32){
            jump++;
            y--;
    }
    else
            jump = 0;
    p[in].pic_sel = 200;
}

//************** fall ************************
```

```
if(!downBlock && jump==0){
    moveFall(&p[in]);
}


crater_num = crater_num_next;
crater_num_next = 0;

 bomb_change()；

//=============================================
if (pressPointer && wi.tilt.y != 90 && wi.tilt.y != 180 && wi.tilt.y != 0){
    ptr.angle = -wi.tilt.y + 90;

}

//=============================================

if (pressBomb){
    strength++;
    printf("Strength = %f", strength);
    b.x = p[in].x;
    b.y = p[in].y;
    if (p[in].dir == LEFT){
            b.diffx = -strength*cos(ptr.angle*PI/180)/100;
            b.diffy = -strength*sin(ptr.angle*PI/180)/100;
    }
    else if (p[in].dir == RIGHT){
            b.diffx = strength*cos(ptr.angle*PI/180)/100;
            b.diffy = -strength*sin(ptr.angle*PI/180)/100;
```

```
    }
    if (b.type == TIMER){
        b.diffx = 0;
        b.diffy = 0;
    }
}

if ((wi.axis.y >= 240) || (wi.axis.z >= 240) || (wi.axis.x >= 240))
    tossFlag = 1;
if (strength > 0 && tossFlag){
    if (b.x < 0){
        b.diffx = abs(b.diffx);
    }
    if (b.x > 640){
        b.diffx = -abs(b.diffx);
    }
    if (b.y < 0 ){
        b.diffy = abs(b.diffy);
    }
    if (b.y > 480){
        b.diffy = -abs(b.diffy);
    }


    if (count >= 100 && count < 190){
        wi.rumble = 1;
        exp_eff = (count-90);
        g_stall=1;
    }
```

```
        if (count >= 190){
            wi.rumble = 0;
            for (i = b.y - 32; i < b.y + 32; i++){
                for (j = b.x - 32; j < b.x +32; j++){
                    if ((j-b.x)*(j-b.x) + (i-b.y)*(i-b.y) < 1024)
                        map[i][j] = 0;
                }
            }
            tossFlag = 0;
            b.type = NONE;
            g_stall=0;
            exp_eff=9;
            count = 0;
            strength = 0;
            timing = 0;
            bomb_used++;
            swapPlayer(&in,&out);
            crater_num_next = bomb_used;
        }

        if(bombLeftBlock || bombRightBlock || bombUpBlock ||
bombDownBlock){
            if (b.type == MISSLE && count <100)
                count = 100;
            if (bombRightBlock && !bombLeftBlock &&   b.diffx >0 ){
                b.diffx = -abs(b.diffx);
            }
            if (bombLeftBlock   && !bombRightBlock &&   b.diffx <0 ){
                b.diffx = abs(b.diffx);
```

```
            }
            if (bombDownBlock    && !bombUpBlock &&    b.diffx >=0 ){
                b.diffy = -abs(b.diffy)/3;
                b.diffx = b.diffx*0.7;

            }
            if (bombUpBlock && !bombDownBlock && b.diffy < 0){
                b.diffy = abs(b.diffy);
            }
            count++;
            if(b.type == TIMER)
                b.diffy = 0;
    }

    if (g_stall != 1){
            b.x += b.diffx;
            b.y += b.diffy;
            b.diffy += acc;
    }

    if (b.type == TIMER && count < 100){
            timing++;
            count = timing/3;
    }

}
write_message();
usleep(8000);

game_over();
```

```
            if (press_home || blood1<=0 || blood2<=0 || crater_num>=20)
                break;
        }
    }
return 0;
}
```

object.h

```
#ifndef OBJECT_
#define OBJECT_

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define PI 3.1415927
enum dir{RIGHT, LEFT};
enum bomb_type{NONE, GRENADE, MISSLE, TIMER};

#include <stdio.h>
#include <stdlib.h>



typedef struct player {
	int x;
	int y;
	int dir;
	int pic_sel;
	int messageH;
	int messageV;
} Player;

typedef struct bomb {
	int x;
```

```
        int y;
        double diffx;
        double diffy;
        int type;
        int messageH;
        int messageV;
} Bomb;

typedef struct pointer{
        double angle;
        int h;
        int y;
        int message;
} Pointer;


Player makePlayer(int x, int y, int dir);
Bomb makeBomb(int x, int y, int type);
Pointer makePointer(double angle);




Player makePlayer(int x, int y, int dir, int pic_sel){
        Player temp;
        temp.x = x;
        temp.y = y;
        temp.dir = dir;
        temp.pic_sel = pic_sel;
```

```
        temp.messageH = 0;
        temp.messageV = 0;
        return temp;
}

Bomb makeBomb(int x, int y, int type){
        Bomb temp;
        temp.x = x;
        temp.y = y;
        temp.type = type;
        temp.messageH = 0;
        temp.messageV = 0;
        return temp;
}

Pointer makePointer(double angle){
        Pointer temp;
        temp.angle = angle;
        temp.h = cos(abs(angle)*PI/180);
        temp.v = sin(abs(angle)*PI/180);
        temp.message = 0;
        return temp;
}

#endif
```

connect.h

```
#ifndef GLOBAL_H_
#define GLOBAL_H_

wiimote_t wi;
vga_led_arg_t vla;

static const char filename[] = "/dev/vga_led";
static unsigned char message[16];

void connect_wii(){
	wiimote_connect(&wi, "0C:FC:83:24:CD:7E");
	wi.mode.acc = 1;		// enable accelerator
}

void update_wii(){
	wiimote_update(&wi);
}

void press_wii(){

	if (wi.keys.b == 1)		// start tilt
		press_pointer = 1;
	else
		press_pointer = 0;

	 if (wi.keys.left == 1)		// LEFT
		press_left = 1;
	else
```

```
        press_left = 0;

     if (wi.keys.right == 1)                  // RIGHT
        press_right = 1;
    else
        press_right = 0;

     if (wi.keys.a == 1)                 // Issue Bomb
        press_bomb = 1;
    else
        press_bomb = 0;

    if (wi.keys.up == 1)                  // Jump
        press_jump = 1;
    else
        press_jump = 0;

    if (wi.keys.plus)               // Change_bomb
        press_change_b = 1;
    else
        press_change_b = 0;


    if (wi.keys.minus)               // Change_bomb_bar
        press_change_b_bar = 1;
    else
        press_change_b_bar = 0;


}
```

```
void connect_vga(){
	if ( (vga_led_fd = open(filename, O_RDWR)) == -1) {
		fprintf(stderr, "could not open %s\n", filename);
		return -1;
	}
	printf("initial state: ");
	print_segment_info();

	printf("current state: ");
	print_segment_info();
}

#endif
```

map.h

```
#ifndef MAP_
#define MAP_

void map_init(){
   for(n=0;n<480;n++){
      for(m=0;m<640;m++){
         map[n][m] = ((m >= 0 && m < 48 && n >= 160 && n < 176)||
                  (m >= 592 && m < 624 && n >= 160 && n < 176)||
                  (m >= 192 && m < 208 && n >= 208 && n < 224)||
                  (m >= 464 && m < 480 && n >= 208 && n < 224)||
                  (m >= 176 && m < 528 && n >= 352 && n < 368)||
                  (m >= 144 && m < 560 && n >= 368 && n < 384)||
                  (m >= 80 && m < 640 && n >= 384 && n < 416)||
                  (m >= 160 && m < 336 && n >= 416 && n < 432)||
                  (m >= 416 && m < 640 && n >= 416 && n < 432)||
                  (m >= 176 && m < 304 && n >= 432 && n < 448)||
                  (m >= 448 && m < 640 && n >= 432 && n < 448)||
                  (m >= 576 && m < 640 && n >= 448 && n < 464)||

                  (m >= 560 && m < 640 && n >= 96 && n < 112)||
                  (m >= 0 && m < 48 && n >= 144 && n < 160)||
                  (m >= 592 && m < 624 && n >= 144 && n < 160)||
                  (m >= 48 && m < 160 && n >= 160 && n < 176)||
                  (m >= 512 && m < 592 && n >= 160 && n < 176)||
                  (m >= 192 && m < 208 && n >= 192 && n < 208)||
                  (m >= 464 && m < 480 && n >= 192 && n < 208)||
                  (m >= 208 && m < 272 && n >= 208 && n < 224)||
```

```
                (m >= 384 && m < 464 && n >= 208 && n < 224)||
                (m >= 272 && m < 384 && n >= 272 && n < 288)||
                (m >= 176 && m < 528 && n >= 336 && n < 352)||
                (m >= 144 && m < 176 && n >= 352 && n < 368)||
                (m >= 528 && m < 560 && n >= 352 && n < 368)||
                (m >= 48 && m < 144 && n >= 368 && n < 384)||
                (m >= 560 && m < 640 && n >= 368 && n < 384));

          map[n][m] += 2*
       (((m >= 0 && m < 16 && n >= 320 && n < 336)||
          (m >= 0 && m < 32 && n >= 336 && n < 352)||
          (m >= 0 && m < 48 && n >= 352 && n < 384)||
          (m >= 0 && m < 80 && n >= 384 && n < 416)||
          (m >= 0 && m < 160 && n >= 416 && n < 432)||
          (m >= 336 && m < 416 && n >= 416 && n < 432)||
          (m >= 0 && m < 176 && n >= 432 && n < 448)||
          (m >= 304 && m < 448 && n >= 432 && n < 448)||
          (m >= 0 && m < 576 && n >= 448 && n < 464)||
          (m >= 0 && m < 640 && n >= 464 && n < 480))
         );
      }
   }
}

void crater_rst(){
   for(n=0;n<32;n++){

      message[3] = 1024;
      message[2] = 0;
      message[1] = n*4;
```

```
        message[0] = 0;

        write_segments(message);
        usleep(80);
    }
}

#endif
```

varibale.h

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int pressPointer = 0;
int pressLeft = 0;
int pressRight = 0;
int pressBomb = 0;
int pressJump = 0;
int pressChangeBomb = 0;

int leftBlock = 0;
int rightBlock = 0;
int upBlock = 0;
int downBlock = 0;
int bombLeftBlock = 0;
int bombRightBlock = 0;
int bombUpBlock = 0;
int bombDownBlock = 0;

int h_player1 = 32+21+10;
int v_player1 = 335-11+21;
int h_player2 = 576+21;
int v_player2 = 335-11+21;
int h_grenade=0, v_grenade=0;
int pointer = 0;
int flag;                    //1: player1, 0: player2
```

```
int flag_next;

double diffx_grenade, diffy_grenade;
int pp =0, pp2 =0;
int pic_sel =200, pic_sel2 = 200;
int direction;

int direction2;

int ko;
int jump;
int left_blocked, g_left_blocked;
int right_blocked, g_right_blocked;
int up_blocked, g_up_blocked;
int down_blocked, down_blocked2, g_down_blocked;
int ready_rebound;
double acc = 0.0625;

int         g_stall=0;
int         exp_eff=9;
double angle = 0;
int point_eabi = 0;
double strength;
int crater_num = 0;
int crater_num_next=0;
int crater_num_delay = 0;
int exp_delay = 0;
int bomb_used =0;
int toss_flag = 0;
char map[480][640];//[vcount][hcount]
```

```
int m,n,i,j;
int timeOut = 31*150;

int bounce = 0;
int count;
int blood1 = 63;
int blood2 = 63;

#endif
void resetBlock(){
	int leftBlock = 0;
	int rightBlock = 0;
	int upBlock = 0;
	int downBlock = 0;
	int bombLeftBlock = 0;
	int bombRightBlock = 0;
	int bombUpBlock = 0;
	int bombDownBlock = 0;
}
```

bomb.h

```
#ifndef control_bomb_h
#define control_bomb_h

void bomb_change(){

	if (bomb_change_delay>30){
		if (press_change_b){
			bomb_type = bomb_type +1;
			x_grenade = x;
			y_grenade = y;
			if (bomb_type >= 4)
				bomb_type = 1;
		}

		if (press_change_b_bar){
			bomb_type = bomb_type -1;
			x_grenade = x;
			y_grenade = y;
			if (bomb_type <= 0)
				bomb_type = 3;
		}
		bomb_change_delay = 0;

	}

}
```

```
void press_bomb_pre(){
	if (press_bomb){
		strength++;
		printf("Strength = %f", strength);
		x_grenade = x;
		y_grenade = y;
		if (direction == LEFT){
			diffx_grenade = -strength*cos(angle*PI/180)/100;
			diffy_grenade = -strength*sin(angle*PI/180)/100;
		}
		else if (direction == RIGHT){
			diffx_grenade = strength*cos(angle*PI/180)/100;
			diffy_grenade = -strength*sin(angle*PI/180)/100;
		}
		if (bomb_type == TIMER){
			diffx_grenade=0;
			diffy_grenade=0;
		}
	}

	if ((wi.axis.y >= 240) || (wi.axis.z >= 240) || (wi.axis.x >= 240))
		toss_flag = 1;
	if (strength!=0 && toss_flag){


		if (x_grenade<0){
			diffx_grenade = abs(diffx_grenade);
		}
		if (x_grenade>640){
```

```
                diffx_grenade = -abs(diffx_grenade);
            }

            if (y_grenade < 0 ){
                diffy_grenade = abs(diffy_grenade);
            }
            if (y_grenade > 480){
                diffy_grenade = -abs(diffy_grenade);
            }



            if (count >= 100 && count <190){
                wi.rumble = 1;
                exp_eff = (count-90);
                g_stall=1;
            }
        }
}
void press_bomb_post(){
        if (count >= 190){
            wi.rumble = 0;
            for (i = y_grenade - 32; i < y_grenade + 32; i++){
                for (j = x_grenade - 32; j < x_grenade +32; j++){
                    if ((j-x_grenade)*(j-x_grenade) + (i-y_grenade)*(i-y_grenade)
< 1024)
                        map[i][j] = 0;
                }
            }
            toss_flag = 0;
            bomb_type = NONE;
```

```
            point_eabi = 0;
            g_stall=0;
            exp_eff=9;
            count = 0;
            strength = 0;
            time_count = 0;
            bomb_used++;
            flag_next = 1 - flag;
            crater_num_next = bomb_used;
        }
}

void if_bomb_block(){
        for(n=1;n<15;n++) {
            g_up_blocked    += map[y_grenade-7][x_grenade+n-7];
            g_down_blocked += map[y_grenade-7+14][x_grenade+n-7];
        }
        for(n=1;n<14;n++) {
            g_left_blocked    += map[y_grenade-7+n][x_grenade-7];
            g_right_blocked += map[y_grenade-7+n][x_grenade+14];
        }
}

void bomb_bounce(){
        if(g_left_blocked || g_right_blocked || g_up_blocked || g_down_blocked){
            if (bomb_type == MISSLE && count <100)
                count = 100;
            if (g_right_blocked && !g_left_blocked &&    diffx_grenade >0 ){
                diffx_grenade = -abs(diffx_grenade);
            }
```

```
        if (g_left_blocked    && !g_right_blocked &&    diffx_grenade <0 ){
                diffx_grenade = abs(diffx_grenade);
        }
        if (g_down_blocked    && !g_up_blocked &&    diffy_grenade >=0 ){
                diffy_grenade = -abs(diffy_grenade)/3;
                diffx_grenade = diffx_grenade*0.7;
    //if (diffx_grenade>=0.001)
    //


        }
        if (g_up_blocked       && !g_down_blocked &&    diffy_grenade <0 ){
                diffy_grenade = abs(diffy_grenade);
        }
        count++;
        if( bomb_type == TIMER)
                diffy_grenade = 0;
}



if (g_stall != 1){
        x_grenade += diffx_grenade;
        y_grenade += diffy_grenade;
        diffy_grenade += acc;
}




if (bomb_type == TIMER && count < 100){
        time_count++;
        count = time_count/3;
```

```
        }
}
#endif
```

move.h

```
#ifndef MOVE_
#define MOVE_

#include <stdio.h>

void stand(Player *pr);
void moveLeft(Player *pr);
void moveLeftBlock(Player *pr);
void moveRight(Player *pr);
void moveRightBlock(Player *pr);
void moveJump(Player *pr);
void swapPlayer(int *in, int *out);



void stand(Player *pr){
	(*pr).pic_sel = 0;
	if ((*pr).pic_sel >= 200)
		(*pr).pic_sel = 0;
}
void moveLeft(Player *pr){
	(*pr).dir = LEFT;
	(*pr).pic_sel += 7;
	if ((*pr).pic_sel >= 500)
		(*pr).pic_sel = 200;
	(*pr).x--;
}
void moveLeftBlock(Player *pr){
```

```
	(*pr).dir = LEFT;
	(*pr).pic_sel += 7;
	if ((*pr).pic_sel >= 500)
		(*pr).pic_sel = 200;
}
void moveRIGHT(Player *pr){
	(*pr).dir = RIGHT;
	(*pr).pic_sel += 7;
	if ((*pr).pic_sel >= 500)
		(*pr).pic_sel = 200;
	(*pr).x++;
}
void moveRightBlock(Player *pr){
	(*pr).dir = RIGHT;
	(*pr).pic_sel += 7;
	if ((*pr).pic_sel >= 500)
		(*pr).pic_sel = 200;
}

void moveFall(Player *pr){
	(*pr).y++;
	(*pr).pic_sel = 200;
}
void swapPlayer(int *in, int *out){
	int temp;
	temp = *in;
	*in = *out;
	*out = temp;
}
```

reset.h

```
#ifndef RESET_H_
#define RESET_H_

void reset_game(){

   message[15] = 0;
   write_segments(message);
   blood1 = 63;
   blood2 = 63;
   flag = 1;
   timeOut= 31*150;
   strength = 0;
   h_player1 = 32+21+10;
   v_player1 = 335-11+21;
   h_player2 = 576+21;
   v_player2 = 335-11+21;
   crater_num_next = 0;
   bomb_type = NONE;
   count = 0;
   bomb_used = 0;
   toss_flag = 0;
   g_stall=0;
   exp_eff=9;
   bounce = 0;
   count = 0;
   strength = 0;
   time_count = 0;
```

```
    message[15] = 0;
    write_segments(message);
}

void game_over(){
    while (blood1<=0 || blood2<=0 || crater_num>=20){
        update_wii();
        press_wii();
        if (blood1>blood2){
            ko = 2;
            v_player2 = v_player2%1024 + (v_player2/32768)*32768 + 6*4096;
            message[9] = v_player2/256;
        }
        else if (blood2>blood1){
            ko = 1;
            v_player1 = v_player1%1024 + (v_player1/32768)*32768 + 6*4096;
                                    message[5] = v_player1/256;
        }

        message[15] = 128;

        write_segments(message);
        if (press_home )
            break;
    }

}

#endif
```

message.h

```
#ifndef control_message_h
#define control_message_h

void write_message(){

	p[0].messageH = p[0].x;
	p[0].messageV = p[0].y + p[0].dir*32768 + (p[0].pic_sel/100)*4096;
	p[1].messageH = p[1].x;
	p[1].messageV = p[1].y + p[1].dir*32768 + (p[1].pic_sel/100)*4096;
	b.messageH = b.x + (exp_eff/10)*4096 + b.type*1024;
	b.messageV = b.y + crater_num*1024;
	ptr.message =
(int)(sin(abs(angle)*PI/180))*64+(int)(cos(abs(angle)*PI/180)*63)*64 +
(angle>=0)*4096 + (1-flag)* 8192 + 1*16384;

	int hex7 =(strength/8)*16 + (timeOut/150)*1024;
	message[15] = hex7 / 256;
	message[14] = hex7 % 256;
	message[13] = ptr.message / 256;
	message[12] = ptr.message % 256;
	message[11] = p[0].messageH / 256;
	message[10] = p[0].messageH % 256;
	message[9] = p[0].messageV / 256;
	message[8] = p[0].messageV % 256;
	message[7] = p[1].messageH / 256;
	message[6] = p[1].messageH % 256;
	message[5] = p[1].messageV / 256;
```

```
        message[4] = p[1].messageV % 256;
        message[3] = b.messageH / 256;
        message[2] = b.messageH % 256;
        message[1] = b.messageV / 256 ;
        message[0] = b.messageV % 256;

        write_segments(message);
}

#endif
```

display.h

```
#ifndef WRITE_VGA_
#define WRITE_VGA_

int vga_led_fd;

/* Read and print the segment values */
void print_segment_info() {
	vga_led_arg_t vla;
	int i;

	for (i = 0 ; i < VGA_LED_DIGITS ; i++) {
		vla.digit = i;
		if (ioctl(vga_led_fd, VGA_LED_READ_DIGIT, &vla)){
			perror("ioctl(VGA_LED_READ_DIGIT) failed");
			return;
		}
		printf("%02x ", vla.segments);
	}
	printf("\n");
}

/* Write the contents of the array to the display */
void write_segments(const unsigned char segs[16])
{
	vga_led_arg_t vla;
	int i;
	for (i = 0 ; i < VGA_LED_DIGITS ; i++) {
```

```
            vla.digit = i;
            vla.segments = segs[i];
            if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla)) {
                perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
                return;
            }
        }
}

#endif
```

System Verilog code

VGA_LED_Emulator

```
module VGA_LED_Emulator(
   input logic              clk50, reset,
 input logic [15:0]    hex0,          //V of grenade          hex0[15:10]:the number of crater
hex0[9:0]: cracter_location
                                 hex1,          //hex1[9:0]: H of grenade hex1[15:12]
explosion figure selec
                                 hex2,          //V 0f stand1      hex2[15]:     0:face to
right, 1:face to left
                                                //                          hex2[14:12]
      show the pic of stand1 from #0 to #6
                                 hex3,          //H of stamd1          hex3[15:10] blood
of player1
                                 hex4,          //V 0f stand2      hex4[15]:     0:face to
right, 1:face to left
                                                //                          hex4[14:12]
      show the pic of stand2 from #0 to #6
                                 hex5,          //H 0f stand2      hex5[15:10] blood of
player2
                                 hex6,          //hex6[5:0]: V of pointer; [11:6]: V of
pointer; hex[14]: en
                                 hex7,          //hex7[14:10] time_count 30 secs
hex7[15] time of player selec
                                                //hex7[9:4] strength
```

```
 output logic [7:0]      VGA_R, VGA_G, VGA_B,
 output logic            VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n,

 input logic[7:0]  in_R,in_G,in_B,                              //soil
                                in_R2,in_G2,in_B2,                      //stone
                                in_R3,in_G3,in_B3,                      //grass
                                in_RG,in_GG,in_BG,                      //grenade
                                in_RL01,in_GL01,in_BL01,                //background
                                in_R_stand1,in_G_stand1,in_B_stand1,//stand1
                                in_R_stand2,in_G_stand2,in_B_stand2,//stand2
                                in_R_pointer,in_G_pointer,in_B_pointer,//pointer

      in_R_explosion,in_G_explosion,in_B_explosion,//explosion
                                in_R_head1, in_G_head1, in_B_head1,   //head1
                                in_R_head2, in_G_head2, in_B_head2,   //head2
                                in_R_ko, in_G_ko, in_B_ko, //ko

 output logic[10:0]     hcount,
 output logic[9:0]vcount );
/*
 * 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
 *
 *HCOUNT 1599 0                  1279          1599 0
 *                ________________                ________
 * ___________|      Video        |_____________|   Video
 *
 *
 * |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
 *         _____________________________      _____________
 * |____|          VGA_HS               |____|
```

```
 */


    parameter HACTIVE      = 11'd 1280,
              HFRONT_PORCH = 11'd 32,
              HSYNC        = 11'd 192,
              HBACK_PORCH  = 11'd 96,
              HTOTAL       = HACTIVE + HFRONT_PORCH + HSYNC +
HBACK_PORCH; //1600

    parameter VACTIVE      = 10'd 480,
              VFRONT_PORCH = 10'd 10,
              VSYNC        = 10'd 2,
              VBACK_PORCH  = 10'd 33,
              VTOTAL       = VACTIVE + VFRONT_PORCH + VSYNC +
VBACK_PORCH; //525


    always_ff @(posedge clk50 or posedge reset)
      if (reset)               hcount <= 0;
      else if (endOfLine) hcount <= 0;
      else                     hcount <= hcount + 11'd 1;

    assign endOfLine = hcount == HTOTAL - 1;


    logic                      endOfField;

    always_ff @(posedge clk50 or posedge reset)
      if (reset)               vcount <= 0;
```

```
    else if (endOfLine)
      if (endOfField)   vcount <= 0;
      else              vcount <= vcount + 10'd 1;

  assign endOfField = vcount == VTOTAL - 1;

  // Horizontal sync: from 0x520 to 0x57F
  // 101 0010 0000 to 101 0111 1111
  assign VGA_HS = !( (hcount[10:7] == 4'b1010) & (hcount[6] | hcount[5]));
  assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

  assign VGA_SYNC_n = 1; // For adding sync to video signals; not used for VGA

  // Horizontal active: 0 to 1279     Vertical active: 0 to 479
  // 101 0000 0000  1280              01 1110 0000  480
  // 110 0011 1111  1599              10 0000 1100  524
  assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
            !( vcount[9] | (vcount[8:5] == 4'b1111) );


  assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge

//=============================================================
//Crater
```

```
//============================================================
logic[19:0]    n_of_crater,crater_flag;
logic[19:0][19:0]  crater_location;

assign n_of_crater=hex0[15:10];

logic crater_binary[63:0][63:0];
assign crater_binary[0] =
64'b0000000000000000000000000011111111111000000000000000000000000000;
assign crater_binary[1] =
64'b0000000000000000000000001111111111111111110000000000000000000000;
assign crater_binary[2] =
64'b0000000000000000000011111111111111111111111100000000000000000000;
assign crater_binary[3] =
64'b0000000000000000001111111111111111111111111110000000000000000000;
assign crater_binary[4] =
64'b0000000000000000111111111111111111111111111111000000000000000000;
assign crater_binary[5] =
64'b0000000000000011111111111111111111111111111111100000000000000000;
assign crater_binary[6] =
64'b0000000000000111111111111111111111111111111111110000000000000000;
assign crater_binary[7] =
64'b0000000000001111111111111111111111111111111111111000000000000000;
assign crater_binary[8] =
64'b0000000000011111111111111111111111111111111111111100000000000000;
assign crater_binary[9] =
64'b0000000000111111111111111111111111111111111111111110000000000000;
assign crater_binary[10] =
64'b0000000001111111111111111111111111111111111111111111000000000000;
```

```
assign crater_binary[11] =
64'b0000000011111111111111111111111111111111111111111111111100000000;
assign crater_binary[12] =
64'b0000000111111111111111111111111111111111111111111111111110000000;
assign crater_binary[13] =
64'b0000001111111111111111111111111111111111111111111111111111000000;
assign crater_binary[14] =
64'b0000011111111111111111111111111111111111111111111111111111100000;
assign crater_binary[15] =
64'b0000011111111111111111111111111111111111111111111111111111100000;
assign crater_binary[16] =
64'b0000111111111111111111111111111111111111111111111111111111110000;
assign crater_binary[17] =
64'b0000111111111111111111111111111111111111111111111111111111110000;
assign crater_binary[18] =
64'b0001111111111111111111111111111111111111111111111111111111111000;
assign crater_binary[19] =
64'b0001111111111111111111111111111111111111111111111111111111111000;
assign crater_binary[20] =
64'b0011111111111111111111111111111111111111111111111111111111111100;
assign crater_binary[21] =
64'b0011111111111111111111111111111111111111111111111111111111111100;
assign crater_binary[22] =
64'b0011111111111111111111111111111111111111111111111111111111111100;
assign crater_binary[23] =
64'b0111111111111111111111111111111111111111111111111111111111111110;
assign crater_binary[24] =
64'b0111111111111111111111111111111111111111111111111111111111111110;
assign crater_binary[25] =
64'b0111111111111111111111111111111111111111111111111111111111111110;
```

```
assign crater_binary[26] =
64'b0111111111111111111111111111111111111111111111111111111111111110;
assign crater_binary[27] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[28] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[29] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[30] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[31] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[32] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[33] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[34] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[35] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[36] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[37] =
64'b1111111111111111111111111111111111111111111111111111111111111111;
assign crater_binary[38] =
64'b0111111111111111111111111111111111111111111111111111111111111110;
assign crater_binary[39] =
64'b0111111111111111111111111111111111111111111111111111111111111110;
assign crater_binary[40] =
64'b0111111111111111111111111111111111111111111111111111111111111110;
```

```
assign crater_binary[41] =
64'b0111111111111111111111111111111111111111111111111111111111111110;
assign crater_binary[42] =
64'b0011111111111111111111111111111111111111111111111111111111111100;
assign crater_binary[43] =
64'b0011111111111111111111111111111111111111111111111111111111111100;
assign crater_binary[44] =
64'b0011111111111111111111111111111111111111111111111111111111111100;
assign crater_binary[45] =
64'b0001111111111111111111111111111111111111111111111111111111111000;
assign crater_binary[46] =
64'b0001111111111111111111111111111111111111111111111111111111111000;
assign crater_binary[47] =
64'b0000111111111111111111111111111111111111111111111111111111110000;
assign crater_binary[48] =
64'b0000111111111111111111111111111111111111111111111111111111110000;
assign crater_binary[49] =
64'b0000011111111111111111111111111111111111111111111111111111100000;
assign crater_binary[50] =
64'b0000011111111111111111111111111111111111111111111111111111100000;
assign crater_binary[51] =
64'b0000001111111111111111111111111111111111111111111111111111000000;
assign crater_binary[52] =
64'b0000000111111111111111111111111111111111111111111111111110000000;
assign crater_binary[53] =
64'b0000000011111111111111111111111111111111111111111111111100000000;
assign crater_binary[54] =
64'b0000000001111111111111111111111111111111111111111111111000000000;
assign crater_binary[55] =
64'b0000000000111111111111111111111111111111111111111111110000000000;
```

```
assign crater_binary[56] =
64'b0000000000011111111111111111111111111111111111111111111100000000000;
assign crater_binary[57] =
64'b0000000000001111111111111111111111111111111111111111110000000000000;
assign crater_binary[58] =
64'b0000000000000111111111111111111111111111111111111111100000000000000;
assign crater_binary[59] =
64'b0000000000000011111111111111111111111111111111111111000000000000000;
assign crater_binary[60] =
64'b0000000000000001111111111111111111111111111111111110000000000000000;
assign crater_binary[61] =
64'b0000000000000000011111111111111111111111111111111000000000000000000;
assign crater_binary[62] =
64'b0000000000000000000111111111111111111111111111100000000000000000000;
assign crater_binary[63] =
64'b0000000000000000000000011111111111111111110000000000000000000000000;


generate
        genvar i;
        for (i=0; i<20; i=i+1) begin : crater_eabi
            assign crater_flag[i] =

    crater_binary[hcount[10:1]-crater_location[i][19:10]+31][vcount[9:0]-crater_location[
i][9:0]+31];

    //((hcount[10:1]-cracter_location[i][19:10])*(hcount[10:1]-cracter_location[i][19:10])+

    //(vcount[9:0]-cracter_location[i][9:0])*(vcount[9:0]-cracter_location[i][9:0]) <= 1024);
        end
```

```
endgenerate


always_ff @(posedge clk50 or posedge reset)
        if (reset)                  crater_location = 20'b0;          //Initialize
        else if  (!(|(n_of_crater)==0))    begin

                        crater_location[n_of_crater]<={hex1[9:0], hex0[9:0]};

                        end

//============================================================
//Indicate the range of moving element and ground_block
//============================================================

logic inpointer, inGrenade, inStand1, inSoil, inGrass, inExplosion,
            inHead1, inHead2, inBlood1, inBlood2, inBloodBlank1, inBloodBlank2,
            inStrengh1, inStrengh2, inStrenghBlank1, inStrenghBlank2, inTime1,
            inTime2, inTimeBlank1, inTimeBlank2, inKo;
logic[7:0] display_select;
logic[7:0] blood_red1, blood_green1, blood_blue1, blood_red2, blood_green2,
blood_blue2,
                    time_red, time_green, time_blue, strength_red1, strength_green1,
strength_blue1,
                    strength_red2, strength_green2, strength_blue2;
logic[5:0]    hexh, hexv;
logic[15:0] hexv_stand, hexh_stand;
logic[9:0] differ_h, differ_v;
```

```
//////////////////////////////////pointer//////////////////
assign hexv = hex6[5:0];
assign hexh = hex6[11:6];
assign hexv_stand = hex6[13]? // 1: stand_2, 0: stand_1;
                                    hex4[15:0]:
                                    hex2[15:0];

assign hexh_stand = hex6[13]?
                                    hex5[15:0]:
                                    hex3[15:0];

assign differ_v = hex6[12]?    // 0: down, 1: up;
                                (hexv_stand[9:0]-hexv):
                                (hexv_stand[9:0]+hexv);

assign differ_h = hexv_stand[15]? //0:face to right, 1:face to left
                                (hexh_stand[9:0]-hexh):
                                (hexh_stand[9:0]+hexh);


//////////////////////////////////////////////

assign inBlood1 = ((hcount/2 > 10'd80) && (hcount/2 <= 10'd80 + 2*hex9[5:0]) &&
                                 (vcount >= 10'd50) && (vcount <= 10'd57));
assign inBlood2 = ((hcount/2 > 10'd560 - 2*hex9[11:6]) && (hcount/2 <    10'd560) &&
                                 (vcount >= 10'd50) && (vcount <= 10'd57));
assign inTime1 = ((hcount/2 > 10'd80) && (hcount/2 <= 10'd80 + 3*hex7[14:10]) &&
                               (vcount >= 10'd36) && (vcount <= 10'd43) && !hex7[15]);//time
left 0-90
```

```
assign inTime2 = ((hcount/2 > 10'd560 - 3*hex7[14:10]) && (hcount/2 < 10'd560) &&
                                (vcount >= 10'd36) && (vcount <= 10'd43) && hex7[15]);//time
right
assign inStrengh1 = ((hcount/2 > 10'd80) && (hcount/2 <= 10'd80 + 3*hex7[9:4]) &&
                                (vcount >= 10'd22) && (vcount <= 10'd29)
&& !hex7[15]);//strength left 0-189
assign inStrengh2 = ((hcount/2 > 10'd560 - 3*hex7[9:4]) && (hcount/2 < 10'd560) &&
                                (vcount >= 10'd22) && (vcount <= 10'd29) && hex7[15]);
assign inBloodBlank1 = ((hcount/2 > 10'd80) && (hcount/2 <= 10'd206) &&
                                        (vcount >= 10'd50) && (vcount <= 10'd57));//left blood
0-126
assign inBloodBlank2       = ((hcount/2 > 10'd434) && (hcount/2 < 10'd560) &&
                                        (vcount >= 10'd50) && (vcount <= 10'd57));//right
        blood
assign inTimeBlank1    = ((hcount/2 > 10'd80) && (hcount/2 <= 10'd170) &&
                                        (vcount >= 10'd36) && (vcount <= 10'd43));//left time
assign inTimeBlank2    = ((hcount/2 > 10'd470) && (hcount/2 < 10'd560) &&
                                        (vcount >= 10'd36) && (vcount <= 10'd43));//right time
assign inStrenghBlank1 = ((hcount/2 > 10'd80) && (hcount/2 <= 10'd269) &&
                                          (vcount >= 10'd22) && (vcount <= 10'd29));//left
strength
assign inStrenghBlank2 = ((hcount/2 > 10'd371) && (hcount/2 < 10'd560) &&
                                          (vcount >= 10'd22) && (vcount <= 10'd29));//right
stength
assign inpointer =(!((in_R_pointer== 8'd0) && (in_G_pointer == 8'd0) && (in_B_pointer
== 8'd0))&&
                                (hcount/2 > (differ_h[9:0]-6)) && (hcount/2 <= (differ_h[9:0]+6))
&&
                                (vcount > (differ_v[9:0]-6)) && vcount <= (differ_v[9:0]+6)&&
hex6[14]);
```

```
assign inExplosion =(!((in_R_explosion== 8'd0) && (in_G_explosion == 8'd0) &&
(in_B_explosion == 8'd0))&&
                                (hcount/2 > hex1[9:0]-32) && (hcount/2 <= hex1[9:0]+32)
&&
                                 (vcount > hex0[9:0]-31) && (vcount <= hex0[9:0]+32));

assign inGrenade =(!((in_RG == 8'd0) && (in_GG == 8'd0) && (in_BG == 8'd0))&&
                                (hcount/2 > (hex1[9:0]-7)) && (hcount/2 <= (hex1[9:0]+7)) &&
                                (vcount > (hex0[9:0]-7)) && (vcount <=(hex0[9:0]+7)));

assign inStand1 = (!((in_R_stand1 == 8'd255) && (in_G_stand1 == 8'd255) &&
(in_B_stand1 == 8'd255))&&
                                (hcount/2 > (hex3[9:0]-21) && hcount/2 <= (hex3[9:0]+22) )&&
                                (vcount > (hex2[9:0]-21) && vcount <= (hex2[9:0]+22)));

assign inStand2 = (!((in_R_stand2 == 8'd255) && (in_G_stand2 == 8'd255) &&
(in_B_stand2 == 8'd255))&&
                                (hcount/2 >(hex5[9:0]-21) && hcount/2 <= (hex5[9:0]+22) )&&
                                (vcount > (hex4[9:0]-21) && vcount <= (hex4[9:0]+22)));

assign inSoil =     ((hcount/2 >= 0 && hcount/2 < 640 && vcount >= 384 && vcount <
448)||
                                (hcount/2 >= 96 && hcount/2 < 560 && vcount >= 368 &&
vcount < 384)||
                                (hcount/2 >= 132 && hcount/2 < 528 && vcount >= 352 &&
vcount < 368))&&
```

```
                        (!(|crater_flag));

assign inGrass = ((hcount/2 >= 0 && hcount/2 < 96 && vcount >= 368 && vcount < 384)||
                        (hcount/2 >= 96 && hcount/2 < 132 && vcount >= 352 &&
vcount < 368)||
                        (hcount/2 >= 132 && hcount/2 < 528 && vcount >= 336 &&
vcount < 352)||
                        (hcount/2 >= 528 && hcount/2 < 560 && vcount >= 352 &&
vcount < 368)||
                        (hcount/2 >= 560 && hcount/2 < 640 && vcount >= 368 &&
vcount < 384))&&
                        (!(|crater_flag));

assign inHead1 =    (hcount/2 > 20 && hcount/2 < 76 && vcount >= 20 && vcount < 58);
assign inHead2 =    (hcount/2 > 564 && hcount/2 < 620 && vcount >= 20 && vcount <
58);

assign inKo = (hcount/2 > 160 && hcount/2 < 481 && vcount >= 150 && vcount < 298)&
hex7[15];

//============================================================
//Indicate which element to display
//============================================================

 always_comb begin
```

```
display_select <=

            inpointer?//pointer
        8'b00000001:
            inExplosion?
        8'b00000010:
            inSoil?//soil
        8'b00000011:
            inGrass?//grass
        8'b00000100:
            (hcount/2 >= 0 && hcount/2 < 640 && vcount >= 448 && vcount < 480)?
        //stone
        8'b00000101:
            inBlood1?
        8'b00000110:
            inBlood2?
        8'b00000111:
            inTime1?
        8'b00001000:
            inTime2?
        8'b00001001:
            inStrengh1?
        8'b00001010:
            inStrengh2?
        8'b00001011:
            inBloodBlank1?
        8'b00001110:
            inTimeBlank1?
        8'b00001111:
            inStrenghBlank1?
```

```
        8'b00010000:
                inBloodBlank2?
        8'b00010001:
                inTimeBlank2?
        8'b00010010:
                inStrenghBlank2?
        8'b00010011:
                inGrenade?  //grenade
        8'b00010100:
                inStand1?
        8'b00010101:
                inStand2?
        8'b00010110:
                inHead1?
        8'b00010111:
                inHead2?
        8'b00011000:
               inKo?
        8'b00011001:
                (hcount/2 >= 0 && hcount/2 < 640 && vcount >= 0 && vcount < 480)?
//Background_left01
        8'b00011010:
        8'b00000000;

{blood_red1, blood_green1, blood_blue1} <=
        ((2*hex9[5:0] >= 7'd0) && (2*hex9[5:0] <= 7'd41))? {8'd200, 8'd0, 8'd0}:
        ((2*hex9[5:0] > 7'd42) && (2*hex9[5:0] <= 7'd83))? {8'd200, 8'd180, 8'd0}:
        {8'd0, 8'd180, 8'd0};

{blood_red2, blood_green2, blood_blue2} <=
```

```
        ((2*hex9[11:6] >= 7'd0) && (2*hex9[11:6] <= 7'd41))? {8'd200, 8'd0, 8'd0}:
        ((2*hex9[11:6] > 7'd42) && (2*hex9[11:6] <= 7'd83))? {8'd200, 8'd180, 8'd0}:
        {8'd0, 8'd180, 8'd0};

{time_red, time_green, time_blue} <=
        (3*hex7[14:10] >= 6'd15)? {8'd120, 8'd187, 8'd254}:{8'd180, 8'd0, 8'd0};

{strength_red1,strength_green1,strength_blue1} <=
        ((hcount/2 - 10'd80>= 8'd0) && (hcount/2 - 10'd80 <= 8'd90))? {2*(hcount/2 - 10'd80)
+ 8'd61, 8'd0, 8'd0}:
        {8'd240, 8'd0, 8'd0 };//red 0-89:151-240 green blue 90-189:0-99

{strength_red2,strength_green2,strength_blue2} <=
        ((10'd560 - hcount/2>= 8'd0) && (10'd560 - hcount/2 <= 8'd90))? {2*(10'd560 -
hcount/2) + 8'd61, 8'd0, 8'd0}:
        {8'd240, 8'd0, 8'd0};//red 0-89:151-240 green blue 90-189:0-99

        case (display_select)
        8'b00000001:{VGA_R, VGA_G, VGA_B} = {in_R_pointer, in_G_pointer,
in_B_pointer };          //pointer
        8'b00000010:{VGA_R, VGA_G, VGA_B} = {in_R_explosion, in_G_explosion,
in_B_explosion };//explosion
        8'b00000011:{VGA_R, VGA_G, VGA_B} = {in_R, in_G, in_B };
                //soil
        8'b00000100:{VGA_R, VGA_G, VGA_B} = {in_R3, in_G3, in_B3 };
                //grass
        8'b00000101:{VGA_R, VGA_G, VGA_B} = {in_R2, in_G2, in_B2 };
                //stone
        8'b00000110:{VGA_R, VGA_G, VGA_B} = {blood_red1, blood_green1,
blood_blue1};
```

```
        8'b00000111:{VGA_R, VGA_G, VGA_B} = {blood_red2, blood_green2,
blood_blue2};
        8'b00001000:{VGA_R, VGA_G, VGA_B} = {time_red, time_green,
time_blue };//time1
        8'b00001001:{VGA_R, VGA_G, VGA_B} = {time_red, time_green,
time_blue };//time2
        8'b00001010:{VGA_R, VGA_G, VGA_B} = {strength_red1, strength_green1,
strength_blue1 };//strength1
        8'b00001011:{VGA_R, VGA_G, VGA_B} = {strength_red2, strength_green2,
strength_blue2 };//strength2
        8'b00001110:{VGA_R, VGA_G, VGA_B} = {8'h20, 8'h20, 8'h20 };//bloodblank1
        8'b00001111:{VGA_R, VGA_G, VGA_B} = {8'h20, 8'h20, 8'h20 };//timeblank1
        8'b00010000:{VGA_R, VGA_G, VGA_B} = {8'h20, 8'h20, 8'h20 };//strengthblank1
        8'b00010001:{VGA_R, VGA_G, VGA_B} = {8'h20, 8'h20, 8'h20 };//bloodblank2
        8'b00010010:{VGA_R, VGA_G, VGA_B} = {8'h20, 8'h20, 8'h20 };//timeblank2
    8'b00010011:{VGA_R, VGA_G, VGA_B} = {8'h20, 8'h20, 8'h20 };//strengthblank2
        8'b00010100:{VGA_R, VGA_G, VGA_B} = {in_RG, in_GG, in_BG };
                //grenade
        8'b00010101:{VGA_R, VGA_G, VGA_B} = {in_R_stand1, in_G_stand1,
in_B_stand1};     //stand1
    8'b00010110:{VGA_R, VGA_G, VGA_B} = {in_R_stand2, in_G_stand2, in_B_stand2};
        //stand2
        8'b00010111:{VGA_R, VGA_G, VGA_B} = {in_R_head1, in_G_head1,
in_B_head1};      //head1
    8'b00011000:{VGA_R, VGA_G, VGA_B} = {in_R_head2, in_G_head2, in_B_head2};
        //head2
    8'b00011001:{VGA_R, VGA_G, VGA_B} = {in_R_ko, in_G_ko, in_B_ko};   //ko
        8'b00011010:{VGA_R, VGA_G, VGA_B} = {in_RL01, in_GL01,
in_BL01 };//Background_left01
        8'b00000000:{VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00};//Black
```

```
        endcase

    end


endmodule // VGA_LED_Emulator
```

VGA_LED.sv

```
module VGA_LED(input logic              clk,
               input logic      reset,
               input logic [7:0]   writedata,
               input logic      write,
               input            chipselect,
               input logic [4:0]   address,

               output logic [7:0] VGA_R, VGA_G, VGA_B,
               output logic          VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
               output logic          VGA_SYNC_n,
               output logic [15:0]   hex0_audio, hex1_audio, hex2_audio, hex3_audio,
                                     hex4_audio, hex5_audio, hex6_audio,
hex7_audio);

logic [23:0] pixel,pixel2,pixel3,pixel4,pixel5,pixel6,pixel7,pixel8,pixel9,pixel10,
pixel_pointer_1,
pixel_player1_0,pixel_player1_1,pixel_player1_2,pixel_player1_3,pixel_player1_4,pixel_
player1_5,pixel_player1_fail,
pixel_player2_0,pixel_player2_1,pixel_player2_2,pixel_player2_3,pixel_player2_4,pixel_
player2_5,pixel_player2_fail,                    pixel_explosion_1,
pixel_explosion_2, pixel_explosion_3, pixel_explosion_4, pixel_explosion_5,
      pixel_explosion_6, pixel_explosion_7, pixel_explosion_8,
pixel_explosion_9,pixel_explosion_back, pixel_head1,
pixel_head2,pixel_greande,pixel_missle,pixel_timer, pixel_ko;

logic [16:0] addr,addr2,addr3,addr4,addr5,addr6,addr7,addr8,addr_type,
                                  addr_pointer_1,
                                  addr_player1,
```

```
                addr_player2,
                addr_explosion,
                addr_head1,
                addr_head2,
                addr_ko;

logic [7:0] R,G,B,                          //soil
                R2,B2,G2,                   //stone
                R3,B3,G3,                   //grass
                RL01,BL01,GL01,             //background
                RG,BG,GG,                   //grenade
                R_player1,G_player1,B_player1,//player1
                R_player2,G_player2,B_player2,//player2
                R_pointer_1,G_pointer_1,B_pointer_1,//pointer
                R_explosion, G_explosion, B_explosion, // explosion
                R_head1,G_head1,B_head1,    //head1
                R_head2,G_head2,B_head2,    //head2
                R_type,G_type,B_type,
                R_ko, G_ko, B_ko;

logic [10:0] hcount;
logic [9:0] vcount;
logic [15:0] hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7;
assign hex0_audio = hex0;
assign hex1_audio = hex1;
assign hex2_audio = hex2;
assign hex3_audio = hex3;
assign hex4_audio = hex4;
assign hex5_audio = hex5;
assign hex6_audio = hex6;
```

```
assign hex7_audio = hex7;



soil soil  (.clock(clk), .address(addr), .q(pixel));
stone stone        (.clock(clk), .address(addr2), .q(pixel2));
grass grass        (.clock(clk), .address(addr3), .q(pixel3));
left01 left01  (.clock(clk), .address(addr4), .q(pixel4));
greande greande(.clock(clk), .address(addr5), .q(pixel5));
missle missle(.clock(clk), .address(addr5), .q(pixel9));
timer timer(.clock(clk), .address(addr5), .q(pixel10));

greande greande_tpye(.clock(clk), .address(addr_type), .q(pixel_greande));
missle missle_type(.clock(clk), .address(addr_type), .q(pixel_missle));
timer timer_type(.clock(clk), .address(addr_type), .q(pixel_timer));

pointer pointer_1(.clock(clk), .address(addr_pointer_1), .q(pixel_pointer_1));
player1 player1_0(.clock(clk), .address(addr_player1), .q(pixel_player1_0));
player1_1
player1_1(.clock(clk), .address(addr_player1), .q(pixel_player1_1));
player1_2
player1_2(.clock(clk), .address(addr_player1), .q(pixel_player1_2));
player1_3
player1_3(.clock(clk), .address(addr_player1), .q(pixel_player1_3));
player1_4
player1_4(.clock(clk), .address(addr_player1), .q(pixel_player1_4));
player1_5
player1_5(.clock(clk), .address(addr_player1), .q(pixel_player1_5));
player1_fail
player1_fail(.clock(clk), .address(addr_player1), .q(pixel_player1_fail));
```

```
                player2 player2_0(.clock(clk), .address(addr_player2), .q(pixel_player2_0));
                player2_1
player2_1(.clock(clk), .address(addr_player2), .q(pixel_player2_1));
                player2_2
player2_2(.clock(clk), .address(addr_player2), .q(pixel_player2_2));
                player2_3
player2_3(.clock(clk), .address(addr_player2), .q(pixel_player2_3));
                player2_4
player2_4(.clock(clk), .address(addr_player2), .q(pixel_player2_4));
                player2_5
player2_5(.clock(clk), .address(addr_player2), .q(pixel_player2_5));
                player2_fail
player2_fail(.clock(clk), .address(addr_player2), .q(pixel_player2_fail));

                explosion_1
explosion_1(.clock(clk), .address(addr_explosion), .q(pixel_explosion_1));
                explosion_2
explosion_2(.clock(clk), .address(addr_explosion), .q(pixel_explosion_2));
                explosion_3
explosion_3(.clock(clk), .address(addr_explosion), .q(pixel_explosion_3));
                explosion_4
explosion_4(.clock(clk), .address(addr_explosion), .q(pixel_explosion_4));
                explosion_5
explosion_5(.clock(clk), .address(addr_explosion), .q(pixel_explosion_5));
                explosion_6
explosion_6(.clock(clk), .address(addr_explosion), .q(pixel_explosion_6));
                explosion_7
explosion_7(.clock(clk), .address(addr_explosion), .q(pixel_explosion_7));
                explosion_8
explosion_8(.clock(clk), .address(addr_explosion), .q(pixel_explosion_8));
```

```
        explosion_9
explosion_9(.clock(clk), .address(addr_explosion), .q(pixel_explosion_9));
        explosion_back
explosion_back(.clock(clk), .address(addr_explosion), .q(pixel_explosion_back));

        head_p1 head_p1(.clock(clk), .address(addr_head1), .q(pixel_head1));
        head_p2 head_p2(.clock(clk), .address(addr_head2), .q(pixel_head2));

        ko ko(.clock(clk), .address(addr_ko), .q(pixel_ko));



        ROMdecoder_1 player1_de(.clk(clk),
                                .hcount(hcount), .vcount(vcount),

    .Routput(R_player1), .Goutput(G_player1), .Boutput(B_player1),
                                .ROMdata_0(pixel_player1_0),
                                .ROMdata_1(pixel_player1_1),
                                .ROMdata_2(pixel_player1_2),
                                .ROMdata_3(pixel_player1_3),
                                .ROMdata_4(pixel_player1_4),
                                .ROMdata_5(pixel_player1_5),
                                .ROMdata_6(pixel_player1_fail),

                                .addr(addr_player1),
                                .hexv(hex2), .hexh(hex3));

        ROMdecoder_player2 player2_de(.clk(clk),
                                .hcount(hcount), .vcount(vcount),
```

```
.Routput(R_player2), .Goutput(G_player2), .Boutput(B_player2),
                                        .ROMdata_0(pixel_player2_0),
                                        .ROMdata_1(pixel_player2_1),
                                        .ROMdata_2(pixel_player2_2),
                                        .ROMdata_3(pixel_player2_3),
                                        .ROMdata_4(pixel_player2_4),
                                        .ROMdata_5(pixel_player2_5),
                                       .ROMdata_6(pixel_player2_fail),
                                        .addr(addr_player2),
                                        .hexv(hex4), .hexh(hex5));

        ROMdecoder_pointer ROMdecoder_pointer1(.clk(clk),
                                        .hcount(hcount), .vcount(vcount),

.Routput(R_pointer_1), .Goutput(G_pointer_1), .Boutput(B_pointer_1),
                                        .ROMdata(pixel_pointer_1),

.hex2(hex2), .hex3(hex3), .hex4(hex4), .hex5(hex5), .hex6(hex6),
                                        .addr(addr_pointer_1));

        ROMdecoder_explosion ROMdecoder_explosion_de(.clk(clk),
                                        .hcount(hcount), .vcount(vcount),

.Routput(R_explosion), .Goutput(G_explosion), .Boutput(B_explosion),
                                        .ROMdata_1(pixel_explosion_1),
                                        .ROMdata_2(pixel_explosion_2),
                                        .ROMdata_3(pixel_explosion_3),
                                        .ROMdata_4(pixel_explosion_4),
                                        .ROMdata_5(pixel_explosion_5),
```

```
                                        .ROMdata_6(pixel_explosion_6),
                                        .ROMdata_7(pixel_explosion_7),
                                        .ROMdata_8(pixel_explosion_8),
                                        .ROMdata_9(pixel_explosion_9),
                                        .addr(addr_explosion),
                                        .hexv(hex0),.hexh(hex1));


        ROMdecoder soil_de(             .clk(clk),
                                        .hcount(hcount), .vcount(vcount),

.Routput(R), .Goutput(G), .Boutput(B),
                                        .ROMdata(pixel),
                                        .addr(addr));

        ROMdecoder stone_de(            .clk(clk),
                                        .hcount(hcount), .vcount(vcount),

.Routput(R2), .Goutput(G2), .Boutput(B2),
                                        .ROMdata(pixel2),
                                        .addr(addr2));

ROMdecoder grass_de(            .clk(clk),
                                        .hcount(hcount), .vcount(vcount),

.Routput(R3), .Goutput(G3), .Boutput(B3),
                                        .ROMdata(pixel3),
                                        .addr(addr3));

        ROMdecoder_left left01_de(  .clk(clk),
```

```
                                        .hcount(hcount), .vcount(vcount),

.Routput(RL01), .Goutput(GL01), .Boutput(BL01),
                                        .ROMdata(pixel4),
                                        .addr(addr4));

        ROMdecoder2 grenade(            .clk(clk),
                                        .hcount(hcount), .vcount(vcount),

.Routput(RG), .Goutput(GG), .Boutput(BG),
                                        .ROMdata_G(pixel5),
                                        .ROMdata_M(pixel9),
                                        .ROMdata_T(pixel10),
                                        .addr(addr5),
                                        .hexv(hex0), .hexh(hex1));

        ROMdecoder_head1 head1_de(          .clk(clk),
                                        .hcount(hcount), .vcount(vcount),

.Routput(R_head1), .Goutput(G_head1), .Boutput(B_head1),
                                        .ROMdata(pixel_head1),
                                        .addr(addr_head1));

ROMdecoder_head2 head2_de(              .clk(clk),
                                        .hcount(hcount), .vcount(vcount),

.Routput(R_head2), .Goutput(G_head2), .Boutput(B_head2),
                                        .ROMdata(pixel_head2),
                                        .addr(addr_head2));
```

```
ROMdecoder3 grenade_type(                .clk(clk),
                                                .hcount(hcount), .vcount(vcount),

.Routput(R_type), .Goutput(G_type), .Boutput(B_type),
                                                .ROMdata_G(pixel_greande),
                                                .ROMdata_M(pixel_missle),
                                                .ROMdata_T(pixel_timer),
                                                .addr(addr_type),
                                                .hexv(hex0), .hexh(hex1));

ROMdecoder_ko ko_de(          .clk(clk),
                                                .hcount(hcount), .vcount(vcount),

.Routput(R_ko), .Goutput(G_ko), .Boutput(B_ko),
                                                .ROMdata(pixel_ko),
                                                .addr(addr_ko));

        VGA_LED_Emulator led_emulator(.clk50(clk),
                                                    .in_R(R), .in_G(G), .in_B(B),

.in_R_pointer(R_pointer_1), .in_G_pointer(G_pointer_1), .in_B_pointer(B_pointer_
1),

.in_R2(R2), .in_G2(G2), .in_B2(B2),

.in_R3(R3), .in_G3(G3), .in_B3(B3),

.in_RL01(RL01), .in_GL01(GL01), .in_BL01(BL01),

.in_RG(RG), .in_GG(GG), .in_BG(BG),
```

```
        .in_R_player1(R_player1), .in_G_player1(G_player1), .in_B_player1(B_player1),

        .in_R_player2(R_player2), .in_G_player2(G_player2), .in_B_player2(B_player2),

        .in_R_explosion(R_explosion), .in_G_explosion(G_explosion), .in_B_explosion(B_
explosion),

        .in_R_head1(R_head1), .in_G_head1(G_head1), .in_B_head1(B_head1),

        .in_R_head2(R_head2), .in_G_head2(G_head2), .in_B_head2(B_head2),

        .in_R_ko(R_ko), .in_G_ko(G_ko), .in_B_ko(B_ko),

        .hex0(hex0), .hex1(hex1), .hex2(hex2), .hex3(hex3), .hex4(hex4), .hex5(hex5), .he
x6(hex6), .hex7(hex7),

        .hex8(hex8), .hex9(hex9), .hex10(hex10), .hex11(hex11), .hex12(hex12), .hex13(h
ex13), .hex14(hex14), .hex15(hex15),

        .hcount(hcount), .vcount(vcount), .*);

           always_ff @(posedge clk)
        if (reset) begin

       hex0 <= 16'b0;
       hex1 <= 16'b0;
       hex2 <= 16'b0;
       hex3 <= 16'b0;
       hex4 <= 16'b0;
```

```
        hex5 <= 16'b0;
        hex6 <= 16'b0;
        hex7 <= 16'b0;
          end else if (chipselect && write)
             case (address)
          5'h0 : hex0[7:0] <= writedata;
             5'h1 : hex0[15:8] <= writedata;
          5'h2 : hex1[7:0] <= writedata;
             5'h3 : hex1[15:8] <= writedata;
          5'h4 : hex2[7:0] <= writedata;
             5'h5 : hex2[15:8] <= writedata;
          5'h6 : hex3[7:0] <= writedata;
             5'h7 : hex3[15:8]<= writedata;
          5'h8 : hex4[7:0] <= writedata;
             5'h9 : hex4[15:8]<= writedata;
          5'ha : hex5[7:0] <= writedata;
             5'hb : hex5[15:8]<= writedata;
          5'hc : hex6[7:0] <= writedata;
             5'hd : hex6[15:8]<= writedata;
          5'he : hex7[7:0] <= writedata;
             5'hf : hex7[15:8] <= writedata;
             endcase

endmodule
```

audio_effects.v

```
module audio_effects (
	input	clk,
	input	sample_end,
	input	sample_req,
	output [15:0] audio_output,
	input	[15:0] audio_input,
	input	[3:0]	control,

	  input wire		write,			// avalon_slave_0.write
	  input wire		chipselect,	//			.chipselect
	  input wire [3:0]	address,		//			.address
	  input wire reset,
	  input wire [15:0] hex0, hex1, hex2, hex3, hex4, hex5, hex6, hex7
);
wire [7:0] romdata_1,romdata_2,romdata_3,romdata_4;
reg [7:0] romdata;
reg [15:0]	index;
reg [15:0] last_sample;
reg [15:0] dat;

assign audio_output = dat;
parameter SINE		= 0;
always @* begin
Audio_display <=
		(hex1[15:12] == 4'd1||hex1[15:12] == 4'd2||hex1[15:12] == 4'd3||
		hex1[15:12] == 4'd4||hex1[15:12] == 4'd5||hex1[15:12] == 4'd6||
		hex1[15:12] == 4'd7||hex1[15:12] == 4'd8||hex1[15:12] == 4'd9)?//explosion
	8'd1:
```

```
            //(hex2[14:12]==3'b010||hex4[14:12]==3'b010)?//jump
            ((hex2[14:12]==3'b010||hex2[14:12]==3'b011||hex2[14:12]==3'b100)||
            (hex4[14:12]==3'b010||hex4[14:12]==3'b011||hex4[14:12]==3'b100))?//walk*/
        8'd2:
            (hex7[3]==1'b1)?//1 throw bomb sound of player1 01 for player2 en
        8'd3:
        8'd0;
end
bomb1 bomb1 (
        .address(index),
        .clock(clk),
        .q(romdata_1));
bomb2 bomb2 (
        .address(index),
        .clock(clk),
        .q(romdata_2));
explosions explosion (
        .address(index),
        .clock(clk),
        .q(romdata_3));
walk walk (
        .address(index),
        .clock(clk),
        .q(romdata_4));

reg [7:0] Audio_display;
reg flag;
always @* begin

        case (Audio_display)
```

```
            8'd1 :begin romdata = romdata_3;//explosion
                        flag = 1;
                        end
            8'd2 :begin romdata = romdata_2;//walk
                        flag = 1;
                        end
            8'd3 :begin romdata = romdata_1;//throw
                        flag = 1;
                        end
            8'd0 :begin romdata = 16'b0;
                         flag = 0;
                         end
      endcase
end
always @(posedge clk) begin
            if (reset) index <= 16'd0;
               if (control[SINE]) begin
                        if ((index < 16'd65535) && flag)begin
                             dat <= {8'd0,romdata};
                             index <= index + 16'd1;
                        end
                        else begin
                             if (flag!=1)
                                  index <= 16'd0;
                        end

             end
               else
                        dat <= 16'd0;
          end
```

```
    end
end


endmodule
```