Star Wars

Final Report

CSEE 4840 Embedded System
Spring 2014
Prof. Stephen A. Edwards

Fang Fang (ff2317)
Jiaxuan Shang (JS4361)
Xiao Xiao (xx2180)
Zhenyu Zhu (zz2281)

Project Overview

The project is inspired by the classic Xbox 360 game *Geometry Wars*. The object of the Star Wars is to survive as long as possible and score as high as possible by destroying an ever-increasing swarm of enemies. The player is able to control a spaceship with a game controller.

Some of the asteroids in the original *Geometry Wars* game are turned into invading spaceships. All the spaceships appear randomly on the screen with spawning figures indicating where there will be enemies coming out. The player is able to shoot bullets to destroy enemies chasing after or flying around the avatar. The player is also able to use a limited number of bombs to clear all the enemies on the screen all at once. The avatar loses one life if the it runs into an enemy. The player uses the left joystick to control the movement of the avatar and the right joystick to control the direction of the bullets that are fired automatically.

Project Design

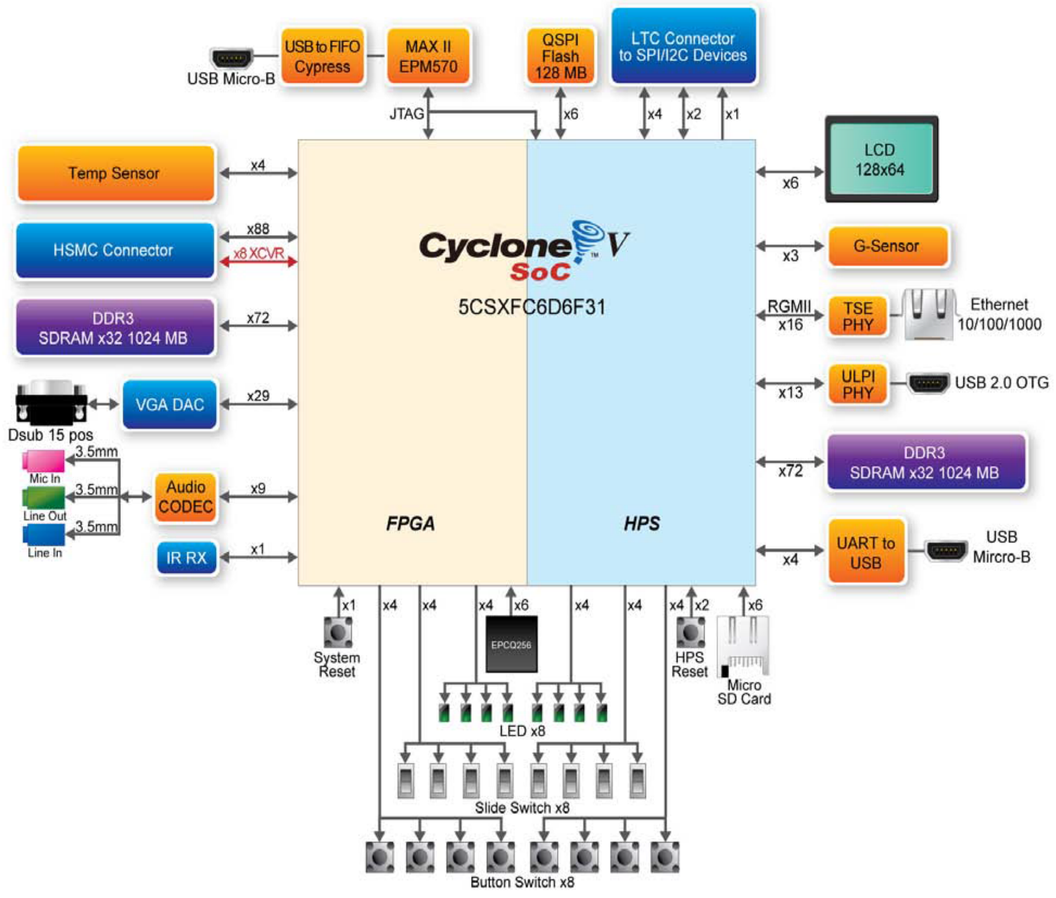


Figure 1 Board block diagram

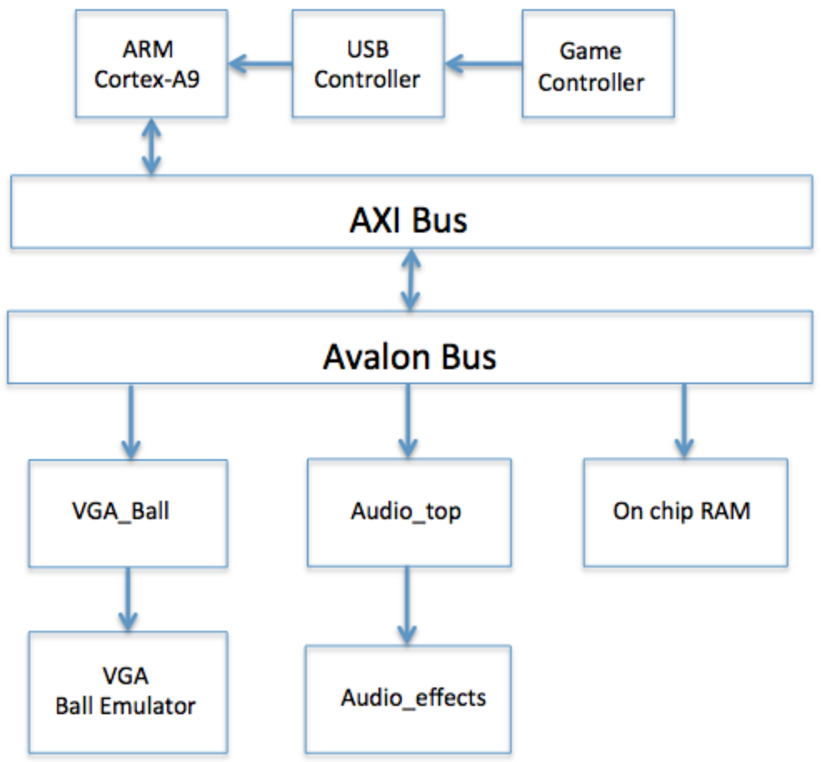


Figure 2 High level architecture of project

The Avalon Bus communicates between the ARM processor and different peripherals. The game controller will be connected to the ARM processor through HPS. We will be using the SDRAM on the HPS for memory to store sprite figures and the sound waveforms. VGA and audio output are connected to the FPGA portion of the board through the Avalon bus. The HPS could talk to FPGA through an AXI bus.

Hardware Design

VGA Top Level Module:

This module (*VGA_BALL.sv*) is the top module receiving data from the avalon bus and sending it to the *VGA_BALL_Emulator* module for VGA screen display. The game supports 60 entities simultaneously on screen and 4 extra information indicating the player status (lifes, bombs, scores, etc). The entity data is stored in this way:

```
        0000_0000__0000_0000_00__0000_0000_00__00__00
        [31:24] ID     [23:14]X         [13:4]Y          [3:0]X_D and Y_D
```

ID is the object type. This is used by the Emulator to determine where to fetched the RGB information. By using 8 bit ID number, the game could easily be extended to support 256 different entities. X and Y are used to tell the position of the each entity on screen. X_D and Y_D are used to store the orientation of each entity. The block diagram for this module is shown below:

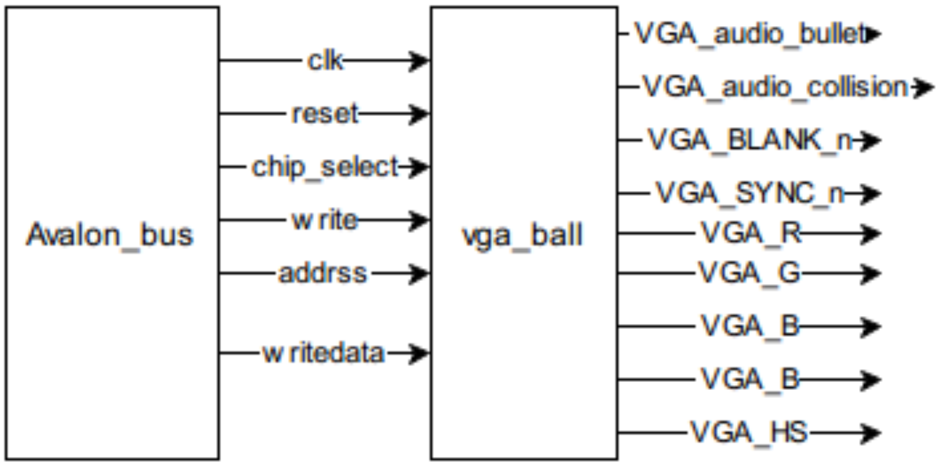


Figure 3 Block diagram for VGA_BALL module

Here, the VGA_BALL module receives data from the Avalon bus, the “write” and “chipselect” signals are used to enable receiving data from the Avalon bus and the “address” and “writedata” are the signals we need to obtain the information of the game. The general state diagram is shown as follows:

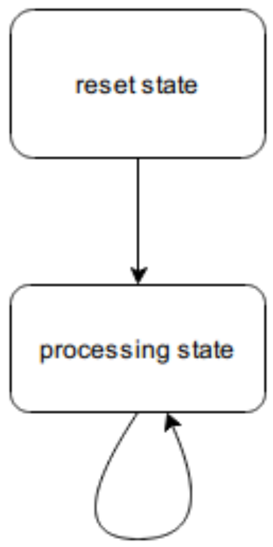


Figure 4 State diagram for VGA_BALL module

In the reset state, we manually reset all the internal signals and the regs used for storing the data. Note that the regs are needed to be treated as memory thus if we need to reset different bits of one element of the reg as different values, it needs to be done in multiple cycles. That's why we need this state instead of simply using the "reset" signal.

In the processing state, it receives 10-bit "writedata" in a total of 256 using 2-reg structure. At the same time, every four of the "writedata" are combined as the needed information into one 32-bit logic: [31:0] logic data_to_emulator_tmp: [id, x, y, direction]. For the same reason as in the reset state, we need 4 cycles to read the data of one object from the reg(memory) and thus the information is stored in the temporal logic data_to_emulator_tmp. After finishing receiving the four parts, it transfers the combined information to "data_to_emulator", which is a 32-bit logic that connects with the VGA_BALL_Emulator to draw all the objects on screen.

The flow chart for the Processing State is shown below:

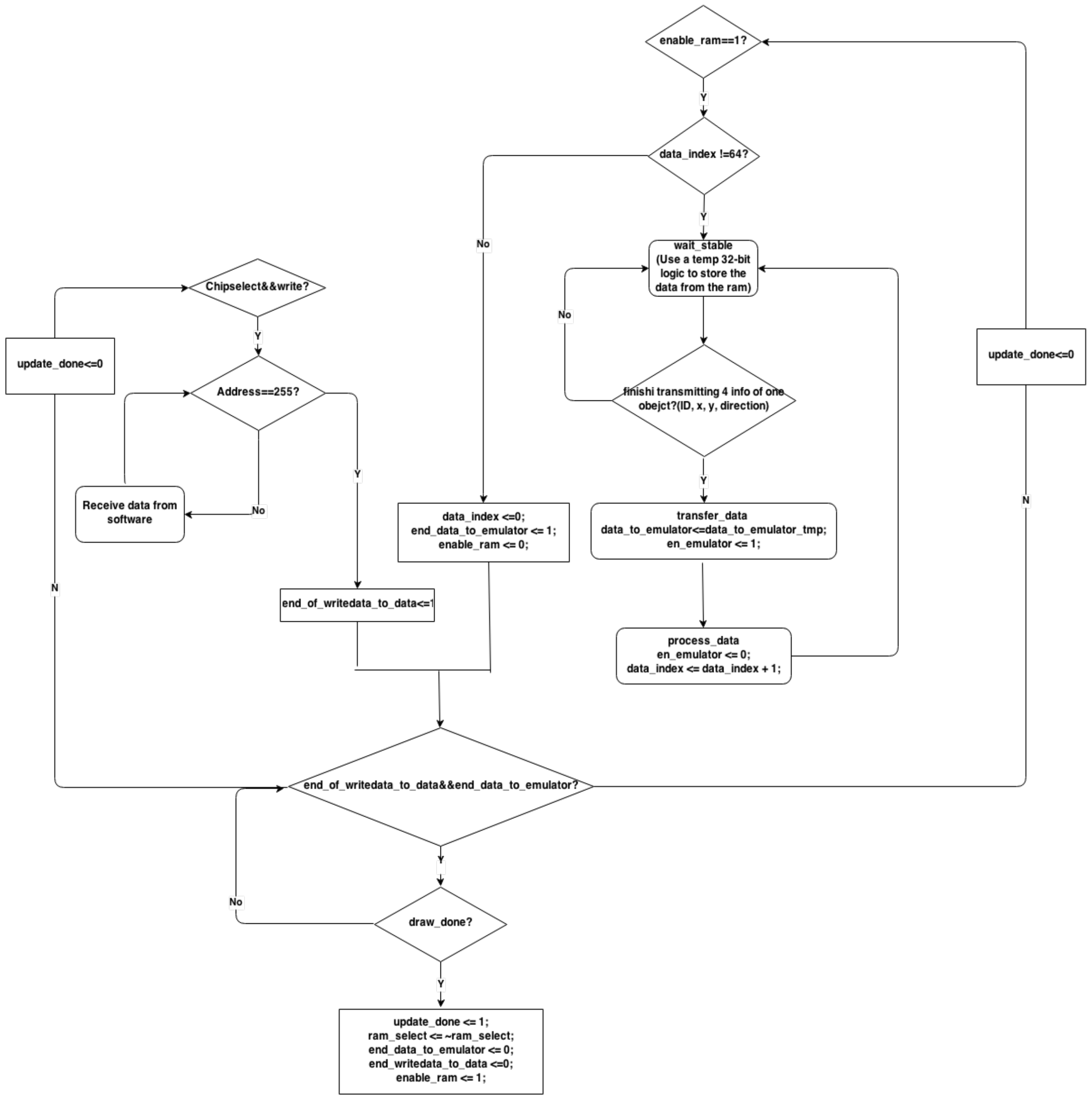
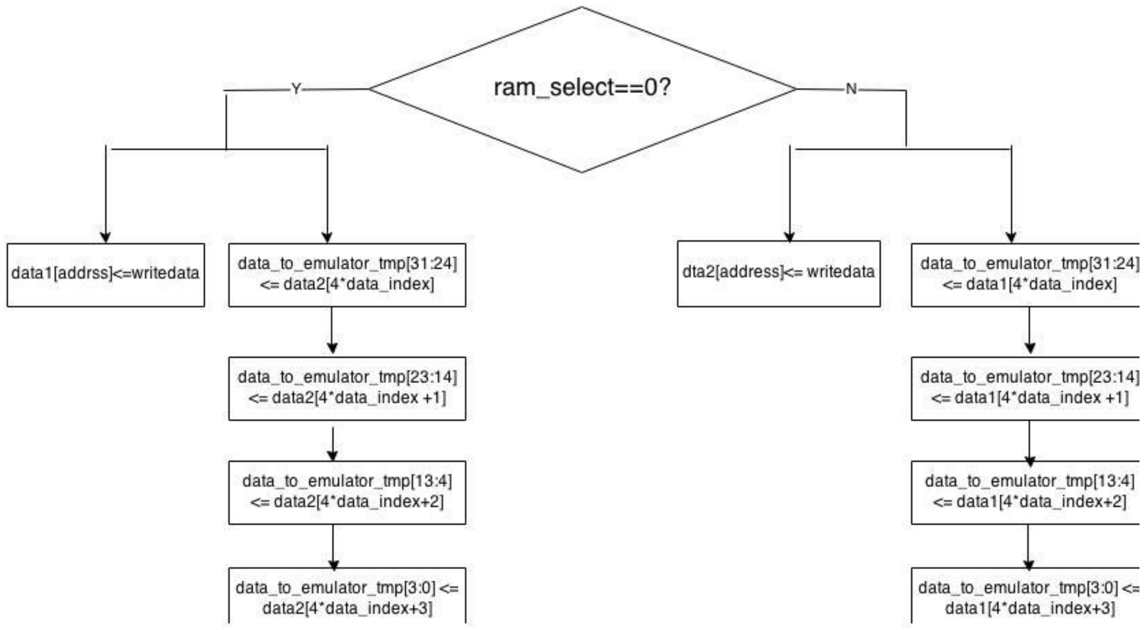

enable_ram==1?
Y
data_index !=64?
No
Y
wait_stable (Use a temp 32-bit logic to store the data from the ram)
No
finishi transmitting 4 info of one obejct?(ID, x, y, direction)
Y
Chipselect&&write?
Y
update_done<=0
Address==255?
Receive data from software
No
Y
N
data_index <=0; end_data_to_emulator <= 1; enable_ram <= 0;
transfer_data data_to_emulator<=data_to_emulator_tmp; en_emulator <= 1;
update_done<=0
N
end_of_writedata_to_data<=1
process_data en_emulator <= 0; data_index <= data_index + 1;
end_of_writedata_to_data&&end_data_to_emulator?
Y
No
draw_done?
Y
update_done <= 1; ram_select <= ~ram_select; end_data_to_emulator <= 0; end_writedata_to_data <=0; enable_ram <= 1;

Figure 5 Processing State flow chart

Seen from this flowchart, the left upper side is for receiving data from the software and store the data into one of the two regs while the right upper side is for fetching the data from the reg and combine into 32-bit logic data_to_emulator_tmp. For the part of receiving data from the software, it would set a flag after finishing receiving 256 messages using 256 cycles and for the part of storing the message into the 32-bit logic, it would also set a flag after combining all the 256 messages into 64 elements using 4*64*2 cycles. Note that the 4 meaning it takes 4 cycles to combine one object, 64 means there are 64 objects to process, and 2 means for the process of every object it needs 2 cycles to complete: one for storing the data, one for adding up the data_index (0 to 63). By this way could we stabilize the data reading from the reg and ensure the

correct timing relates to the VGA_BALL_Emulator (which utilizes 2-ram structure to store the data_to_emulator into the rams and also need 2 cycles for stabilization).
Another thing to mention for this module is that the timing between reading from the software and transmitting the combined data to the emulator. Here we use 2-reg structure to avoid the possible lag if processing the two parts sequentially. One is for reading the data from the software while the other is for transmitting to the emulator at the same time. The flowchart for this part is shown below:


ram_select==0?
Y
N
data1[addrss]<=writedata
data_to_emulator_tmp[31:24] <= data2[4*data_index]
data_to_emulator_tmp[23:14] <= data2[4*data_index +1]
data_to_emulator_tmp[13:4] <= data2[4*data_index+2]
data_to_emulator_tmp[3:0] <= data2[4*data_index+3]
dta2[address]<= writedata
data_to_emulator_tmp[31:24] <= data1[4*data_index]
data_to_emulator_tmp[23:14] <= data1[4*data_index +1]
data_to_emulator_tmp[13:4] <= data1[4*data_index+2]
data_to_emulator_tmp[3:0] <= data1[4*data_index+3]
Figure 6 Flow chart for the mechanism of 2-reg structrue

It could be clearly seen that the ram_select decides who does what at the same time. This signal toggles every time the VGA_BALL_Emulator finishes drawing the whole field (1600*525 cycles).

VGA Display Module:
This module (*VGA_Ball_Emulator.sv*) is responsible for the VGA display on the screen. On the one hand, it communicates with VGA top level module to get the data it needed. On the other hand, it used three line buffers to produce a video signals that will draw sprites on the screen. In order to achieve functions mentioned above, the VGA Display Module does following things:

1. At the beginning, this module is in sleep module, which means until there is input signal (update_done) tells it to work, it will do nothing.
2. After receive the start signal, update_done, this module will start to increase hcount and vcount at the rising edge of clk50..
3. At the same time, if entity information is sent from the VGA top level, this module starts to use ram to store the information. In order to avoid timing issue during drawing sprites,

two rams are used here. One is used to receive data to store incoming information, the other is used to draw sprites on the graph. By using this struct, the read during write behavior can avoid and there is no need to wait the information to draw sprites. There is another signal used here call ram_select. If ram_select is 0, ram b is used to store incoming data and ram a is used to draw sprites. If ram_select is 1, ram a is used to store and ram b is used to draw. ram_select signal will be toggled each time this module finishes its work. By doing this, there will be no two consecutive frames have the same ram_select signal, which maintains the right function of above two rams

4. Meanwhile, three line buffers are used to draw sprites. There are three line buffers called A, B, C in this module. The following graph shows the working sequence for them .

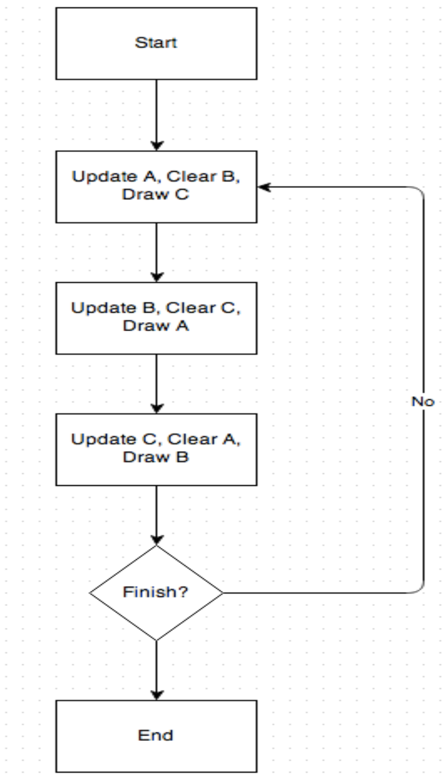
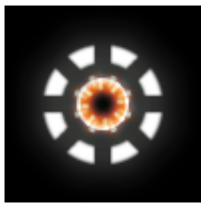

Start
Update A, Clear B, Draw C
Update B, Clear C, Draw A
Update C, Clear A, Draw B
No
Finish?
End

Figure 7 Line buffer's wokring sequence flow chart

At the first time, line buffer A is used to update the information needed to be draw, line buffer B is preparing for next updating by clearing the information in it, and line buffer C is used to draw sprites on the screen. Next time, B is used to update, C is used to prepare for updating and A is used to draw. Then C is used to update, A is used to prepare for updating and B is used to draw. After this, endOfField signal will be checked. This signal indicates whether this module finishes all the drawing on the screen. If endOf Field is 1, then line buffers will go to idle state to wait for another trigger and starts again. If endOfField is 0, then the line buffers will keep working as the way described above.
The line buffer is made of up on chip ram. Since there are 640 pixels in one line and each pixel is controller by 24 bit RGB value, the size of it is 640*24 bits. At each rising edge of clk50, the entity data stored in the ram that is for drawing is fetched to see if it is

on the current working line.If the entity fetched indicates that it should appear on the current updating line buffer, by using the fetched entity data, which include ID, X, Y, X_D and Y_D, RGB data will be read from rom and stored in the right position in updating line buffer. This will keep working until all the entity data in ram is checked. At the same time, at each rising edge of clk50, RGB value stored in drawing buffer will be read to draw sprites according to the value of hcount/2. In addition, RGB value stored in cleaning buffer will be deleted at the rising edge of clk50 according to the value of hcount/2.
Another thing needed to mention is that by using above three line buffer structure, the



overlap issue is solved perfectly. The overlap issue is that each sprite is placed on a black frame as
shown in following graph. When two or more entities meet in the flight, overlap parts will be replaced by the black frame. The solution to this that, before writing RGB value to the updating line buffer, the RGB value is checked. If the RGB value is 0, which means black, the original data in the line buffer will not change by this RGB. By doing this, the overlap issue disappear.

Audio Module:
This module take advantage of the Audio CODEC component in the board. The SoCKit board provides high-quality 24-bit audio via the Analog Devices SSM2603 audio CODEC (Encoder/Decoder). This chip supports microphone-in, line-in, and line-out ports, with a sample rate adjustable from 8 kHz to 96 kHz. In our project, we only use line-out port, and the sample rate we choose is 32kHz. The SSM2603 is controlled via a serial I^2C bus interface, which is connected to pins on the Cyclone V SoC FPGA. For the I^2C protocol, the FPGA is the master and Audio CODEC is the slave in our case. A schematic diagram of the audio is shown in the following figure.

As seen in the figure, there are five different clock signals and two data signals on the digital audio interface. The two wires in the I^2C protocol are labeled SDAT and SCLK for Serial Data and Serial Clock respectively. Data is sent a bit at a time over the SDAT wire, with the separation between bits determined by clock cycles on the SCLK wire.

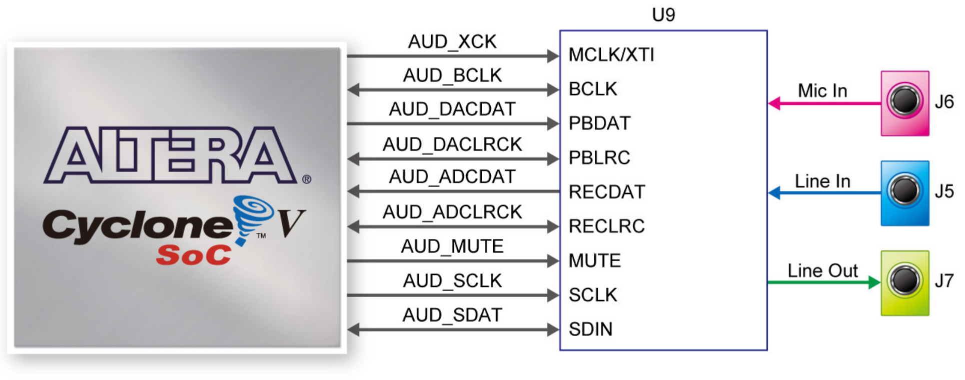


Figure 8 Connections between FPGA and Audio CODEC

The I^{2}C controller in our project acts as a way to send data through configuration interface. The part of sending data is controlled by configuration controller. It steps through different 16-bit data words for the I^{2}C controller one at a time. The transferred 16-bit data words contain address information and specific configuration information. The audio codec organizes its configuration variables into 19 9-bit registers. The first seven bits of the data transmission are the register address, and the last nine bits are the register contents.The detailed audio configuration information for using the SSM2603 codec is searched in its datasheet based on our audio file properties.

As for the clock part, the first clock we have to worry about generating is the master clock MCLK. According to the datasheet, the frequency of this clock should be 12.288 MHz. This frequency cannot be generated by simply dividing the master clock since there is no integer number N such that 50 / N is close enough to 12.288. Therefore, we use the specialty circuits called Phase-Locked Loops (PLLs) contained in Cyclone V. PLLs can generate very precise clock signals. The next clock we have to consider is the bit clock BCLK. According to the datasheet, this clock should be a quarter the frequency of the master clock. We can easily generate this using a frequency divider on the audio clock from the PLL. The last two clocks are RECLRC and PBLRC left right clocks, which tell the codec which of the two stereo audio channels is being accessed. In our case, we only use the left channel by setting the two LRC signals high. These two clocks are 384 times slower than the MCLK frequency. So the clock frequency division is 12.288 MHz / 384 = 32 kHz. In the audio codec driver, the clock is passed in from the PLL and the data is pushed out or read in through shift registers.

All of the modules are included in the top-level (*audio_top.sv*), and this part holds all of the audio control part logic. In order to playout the sound effect, another module (*audio_effects.sv*) contains the testing logic. In this module, the sound is played out by reading .mif data that is stored in the ROM. And the specific enable flag is received from software part (*hello.c, message[244]*) to decide whether to play out the collision sound effect.

Software Design

Vga_ball.c and Vga_ball.h : These files are modified from the vga_led.c and vga_led.h provided by the professor Stephen. The main change is that originally, vga_led.c just supports 8 bit value operations. If the data is more than 8 bits, it will fail to work right. Because of the screen is 640*480, the position of each entity needed to be represented as at least 10 bit number to cover the entire screen.

Usbkeyboard.c and Usbkeyboard.h: These files are modified from the usbkeyboard. c and usbkeyborad.h provided by the professor Stephen. The main change is that originally, usbkeyborad.c is used to detect the usb keyboard. However, since this game is based on the game controller, something needed to be changed to detect game controller. By using lsusb -v command in the terminal. we can get all the information about the game controller that connected to the board. Based on this information, usbkeyborad.c and usbkeyborad.h are changed accordingly.

Hello.c: This is the major part of the software program of the project. This C code is responsible for the bullets and enemy generation and also the movement of all the entities in the game. The array message also stores all the player and game information such as data of the score, life and bomb left.
An overall flow chart of the logic in this c code is as below.

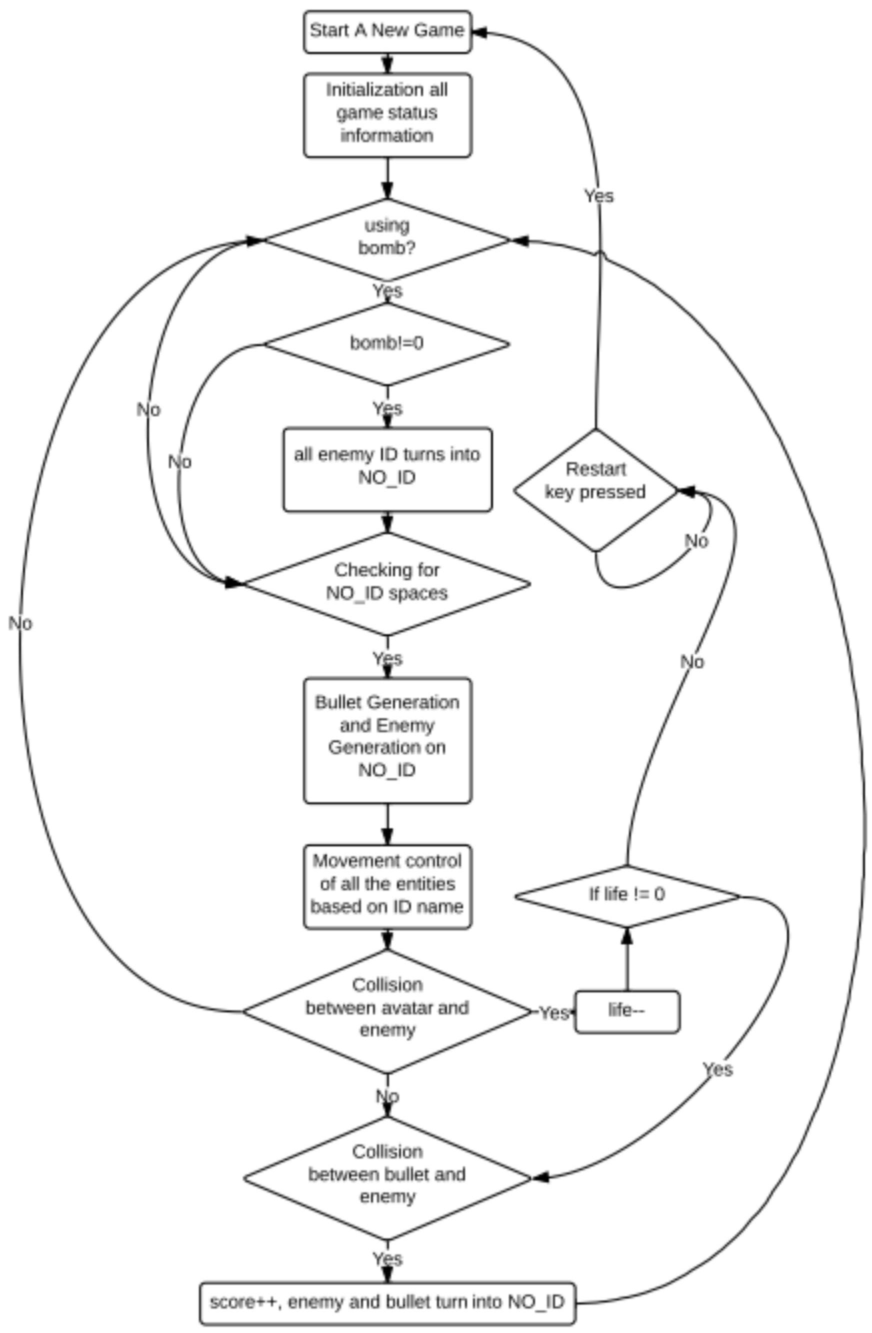

Start A New Game
Initialization all game status information
using bomb?
Yes
bomb!=0
Yes
all enemy ID turns into NO_ID
Checking for NO_ID spaces
Yes
Bullet Generation and Enemy Generation on NO_ID
Movement control of all the entities based on ID name
Collision between avatar and enemy
Yes
life--
If life != 0
No
Restart key pressed
No
Yes
Yes
No
Collision between bullet and enemy
Yes
score++, enemy and bullet turn into NO_ID
No
No
No

Figure 9 Software flow chart

We use a 256-element array to store all the entity information we need for the game. We have a total of 60 entities and with each entity having four data values to store, giving 240 in total. The first four elements are initialized to be the ID of the avatar and the starting point of the avatar. All other elements in the array are initialized to be zero. The last 16 elements are used to store all the player information.

```
static unsigned int message[256]= {SPACESHIP, 320, 240, 0};
```

Here we have the *random number generator* which generates the *x* and *y* coordinates of the spawning entity which will turn into enemies randomly on the screen. The generated *x* and *y* are bounded to be inside the visible game area on the screen by using the modulus 480 and modulus 440.

```
/*random_number_generation*/
unsigned int random_number_generation_x()
{
  unsigned int random_num;
  random_num= rand()%480+1;
  return random_num;

}
unsigned int random_number_generation_y()
{
  unsigned int random_num;
  random_num= rand()%440+1;
  return random_num;
}
```

The a few lines below are part of the c code that does the interaction between the game controller and the movement of the avatar and the direction of the bullets. If the joystick is pushed to a certain direction, the variable used to save the information of the direction will be set to 1. This part is also responsible for the controlling of the shooting direction of the bullets. In total we have eight direction for the avatar and eight directions for the bullets. Two of the buttons are used to restart the game and control the use of the bomb.
This is also the place where we are checking if the player is using a bomb.

```
if (packet.keycode[4] == 0x08 && message[242] != 0 && bomb == 0 && bomb_status < 30)
{
       bomb_status = bomb_status + 1;
       if (bomb_status == 30) {
              bomb_status = 0;
              bomb = 1;
              message[242] = message[242] - 1;
       }
}
if (packet.keycode[4] == 0x20) { /* R2 pressed? start a new game  */
       start = 1;
       game_over = 0;
       message[255] = 0;
}
if (message[255] == 0 && message[240] > 0) {
       if (packet.keycode[3] == 0x4f) {
              down_d = 1;
```

```
        }
        if (packet.keycode[3] == 0xcf) {
                leftdown_d = 1;
        }
}
```

After getting the desired direction where the avatar and bullets are moving to from the controller, we are now ready for the bullets and enemy generation. Since all the entity spaces from 2 to 30 are saved for bullets, we are continuously updating new direction information to the bullets.

```
if (counter_bullet == 15) {
        int i;
        for (i = 4; i < 120; i = i + 4) //The bullet info start from message[4]
        {
                        //generate the id of the bullet
                        if (message[i] == 0) {
                        flag_bullet = 1;
                        message[i] = BULLET; //define the id
                        if (left_d == 1) {
                                message[i + 3] = LEFT;
                        }
                        if (leftup_d == 1) {
                                message[i + 3] = LEFTUP;
                        }
                        …
                        …
                        …
        }
}
```

The bullet counter is introduced to control the speed generation of the bullets. The same technique is also used in the movement control and enemy generation.
Next we are moving to the enemy generation. Enemy are generated with a spawning indicating that there will be an enemy coming out.

After spawning, we continue to the movement control and in this part different enemies will be generated randomly with different features after spawning. Some of the enemies will be flying round and some will be chasing after the spaceship.

Then we will be detecting if there are any collision between the entities and this is where a separate *collision.c* is used to detect the collision by comparing their coordinate difference with the sum of their radius.
There are good collisions and bad collisions, with good ones the bullets destroy the enemy and enemy ID turns into NO_ID indicating it’s a free space to generate new spawnings and the player gets one score increased, with bad collisions the spaceship hits the enemy and one life is lost.

Whenever a life is lost we check if life is zero or not, and with a 0 life we could wait for the restart button to be pushed to start a new game.

Lessons Learned

After one semester's study and work on the SocKit board, we have mastered the architecture of the hardware on FPGA board, the connection of HPS and FPGA, and the importance of Avalon Bus. The skills in both software language (C programming) and hardware language (SystemVerilog) have been greatly increased and continuously improving. Besides, we are familiar with some CAD tools such as Quartus (including Signaltap, Netlist Viewer) to figure out problems.

Resource allocation – Manage to minimize logic utilization.

1. We have learned that in FPGA, if a reg array is defined as logic [31:0] sample [0:59], Quartus will treat it as the ram unless the configuration of Quartus is made to force Quartus to stop this transformation. For a single port memory block, it is essential only to read/write one value from/into the memory at one clock cycle. At first we didn't recognize this problem and wrote different parts of one data into different values in one clock cycle. This makes Quartus confuse and treat reg array as logic. This enormously increased the logic utilization (up to 40%+) since the reg could no longer be memory but only logic and plenty of flip-flops are added to fulfill the requirement. After looking through the generated RTL netlist, we realized this problem and corrected it, then the logic utilization is under 30%.

2. Line buffer to draw the graph. Since we have at most 64 objects to display on screen at one time, by using for loops to draw sprites on screen will use a lot of logics. line buffer (when updating the data from the vga_ball into one ram, use another ram to draw the graph and one for cleaning and preparing for receiving next time) is used to reduce logics. . This part has been described in detail in previous section. Most importantly, what the line buffer should store is the RGB value of every object instead of the copy of 32-bit information of every object. By this way the logic utilization is further scaled down to 17% (currently).

Future Work suggestions:

1. The enemies' characteristics can be more various. For example, they can shoot bullet too. The bullets coming from the enemies have two kinds of colors, black and white. The spaceship that the player controls will be able to change its color to match the color of the incoming bullets to avoid damage and increase energy.
2. Two modes can be added to the game for player to choose, easy and hard.
3. Many other enemy track types can be implemented in software such as circle track.
4. The sound effects could be improved.

Contribution:

Fang Fang (ff2317): VGA_BALL module (VGA_controller) and part of the software
Jiaxuan Shang (js4361): Software for the game logic, image processing
Xiao Xiao (xx2180): Audio module, part of software
Zhenyu Zhu (zz2281): VGA_emulator module, game controller (USB driver modification), part of VGA_BALL module

Reference:

1. SoCKit User Guide
   http://www.rocketboards.org/pub/Documentation/ArrowSoCKitEvaluationBoard/SoCKit_User_manual.pdf
2. SoCKit Getting Started Guide
   http://www.rocketboards.org/pub/Documentation/ArrowSoCKitEvaluationBoard/SoCKit_QSG.pdf
3. Vertex. Final design report. Don Goldin Mark Sullivan
   http://web.mit.edu/6.111/www/f2008/projects/aureus_Project_Final_Report.pdf
4. Exploring the Arrow SoCKit Part VIII - The Audio Codec
   http://zhehaomao.com/blog/fpga/2014/01/15/sockit-8.html

Appendix:

Software:

1. vga_ball.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_ball.h"

#define DRIVER_NAME "vga_ball"

//Information about our device
struct vga_ball_dev {
	struct resource res; //Resource:our registers
	void __iomem *virtbase; // Where registers can be accessed in memory
	 int     directions[256];         ////fffff //JS
} dev;

/*
 * Write a direction of a sigle axis.
 * Asume we have x and y axises.
 * for y, write the value of direction to the address of dev.virtabse+1
 * and change the value of directions[1]
 * for x, write the value of direction to the address of dev.virtabse
 * and change the value of directions[0]
 */
static void write_axis(int axis, int direction)
{
	iowrite16(direction, dev.virtbase + axis*2);
	dev.directions[axis] = direction;
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */

static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
	vga_ball_arg_t vla;

	switch (cmd) {
	case VGA_BALL_WRITE_AXIS:
```

```
            if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                  sizeof(vga_ball_arg_t)))
                  return -EACCES;
            if (vla.axis > 256)
                  return -EINVAL;
            write_axis(vla.axis, vla.direction);
            break;
      case VGA_BALL_READ_AXIS:
            if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                               sizeof(vga_ball_arg_t)))
                  return -EACCES;
            if (vla.axis > 256)
                  return -EINVAL;
            vla.direction = dev.directions[vla.axis];
            if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                             sizeof(vga_ball_arg_t)))
                  return -EACCES;
            break;
      default:
            return -EINVAL;
      }

      return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_ball_fops = {
      .owner       = THIS_MODULE,
      .unlocked_ioctl = vga_ball_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_ball_misc_device = {
      .minor       = MISC_DYNAMIC_MINOR,
      .name        = DRIVER_NAME,
      .fops        = &vga_ball_fops,
};

/*
 * Initialization code: get resources (registers)
 */
static int __init vga_ball_probe(struct platform_device *pdev)
{
      int ret;

      /* Register ourselves as a misc device: creates /dev/vga_ball */
      ret = misc_register(&vga_ball_misc_device);

        /* Get the address of our registers from the device tree */
      ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
      if (ret) {
            ret = -ENOENT;
            goto out_deregister;
      }
```

```
	 /* Make sure we can use these registers */
	if (request_mem_region(dev.res.start, resource_size(&dev.res),
			       DRIVER_NAME) == NULL) {
		ret = -EBUSY;
		goto out_deregister;
	}

	/* Arrange access to our registers */
	dev.virtbase = of_iomap(pdev->dev.of_node, 0);
	if (dev.virtbase == NULL) {
		ret = -ENOMEM;
		goto out_release_mem_region;
	}

	return 0;

	out_release_mem_region:
		release_mem_region(dev.res.start, resource_size(&dev.res));
	out_deregister:
		misc_deregister(&vga_ball_misc_device);
		return ret;
}

/* Clean-up code: release resources */
static int vga_ball_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&vga_ball_misc_device);
	return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
	{ .compatible = "altr,vga_ball" },
	{},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_ball_driver = {
	.driver	= {
		.name	= DRIVER_NAME,
		.owner	= THIS_MODULE,
		.of_match_table = of_match_ptr(vga_ball_of_match),
	},
	.remove	= __exit_p(vga_ball_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
```

```
{
	pr_info(DRIVER_NAME ": init\n");
	return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

/* Called when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
	platform_driver_unregister(&vga_ball_driver);
	pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Zhenyu Zhu, Columbia University");
MODULE_DESCRIPTION("VGA BALL Emulator");
```

2. usbkeyboad.c

```
#include "usbkeyboard.h"

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

/* References on libusb 1.0 and the USB HID/keyboard protocol
 *
 * http://libusb.org
 * http://www.dreamincode.net/forums/topic/148707-introduction-to-using-libusb-10/
 * http://www.usb.org/developers/devclass_docs/HID1_11.pdf
 * http://www.usb.org/developers/devclass_docs/Hut1_11.pdf
 */

/*
 * Find and return a USB keyboard device or NULL if not found
 * The argument con
 *
 */
struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address) {
  libusb_device **devs;
  struct libusb_device_handle *keyboard = NULL;
  struct libusb_device_descriptor desc;
  ssize_t num_devs, d;
  uint8_t i, k;

  /* Start the library */
  if ( libusb_init(NULL) < 0 ) {
    fprintf(stderr, "Error: libusb_init failed\n");
    exit(1);
  }

  /* Enumerate all the attached USB devices */
  if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
    fprintf(stderr, "Error: libusb_get_device_list failed\n");
    exit(1);
  }

  /* Look at each device, remembering the first HID device that speaks
     the keyboard protocol */

  for (d = 0 ; d < num_devs ; d++) {
    libusb_device *dev = devs[d];
    if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
      fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
      exit(1);
    }

    if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE) {
      struct libusb_config_descriptor *config;
      libusb_get_config_descriptor(dev, 0, &config);
      for (i = 0 ; i < config->bNumInterfaces ; i++)
       for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {
         const struct libusb_interface_descriptor *inter =
           config->interface[i].altsetting + k ;
         if ( inter->bInterfaceClass == LIBUSB_CLASS_HID &&
              inter->bInterfaceProtocol == USB_HID_KEYBOARD_PROTOCOL) {
           int r;
           if ((r = libusb_open(dev, &keyboard)) != 0) {
             fprintf(stderr, "Error: libusb_open failed: %d\n", r);
             exit(1);
           }
           if (libusb_kernel_driver_active(keyboard,i))
             libusb_detach_kernel_driver(keyboard, i);
           libusb_set_auto_detach_kernel_driver(keyboard, i);
           if ((r = libusb_claim_interface(keyboard, i)) != 0) {
             fprintf(stderr, "Error: libusb_claim_interface failed: %d\n", r);
             exit(1);
           }
           *endpoint_address = inter->endpoint[0].bEndpointAddress;
           goto found;
         }
       }
    }
  }

 found:
  libusb_free_device_list(devs, 1);

  return keyboard;
}
```

3. usbkeyboard.h

```
#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include <libusb-1.0/libusb.h>
```

```
#define USB_HID_KEYBOARD_PROTOCOL 0

/* Modifier bits */
#define USB_LCTRL  (1 << 0)
#define USB_LSHIFT (1 << 1)
#define USB_LALT   (1 << 2)
#define USB_LGUI   (1 << 3)
#define USB_RCTRL  (1 << 4)
#define USB_RSHIFT (1 << 5)
#define USB_RALT   (1 << 6)
#define USB_RGUI   (1 << 7)

struct usb_keyboard_packet {
  uint8_t modifiers;
  uint8_t reserved;
  uint8_t keycode[6];
};

/* Find and open a USB keyboard device.  Argument should point to
   space to store an endpoint address.  Returns NULL if no keyboard
   device was found. */
extern struct libusb_device_handle *openkeyboard(uint8_t *);

#endif
```

4. hello.c

```
/*
 * Userspace program that communicates with the vga_ball device driver primarily through ioctls
 * Control the game logic, including the infomation generation of all obejcts, movement, collision
detection
 * Zhenyu Zhu Fang Fang Jiaxuan Shang Xiao Xiao
 * StarWars
 * CSEE4840 Embedded Systems Design Spring 2014
 * Columbia University
 */

# include < stdio.h >
# include < stdlib.h >
# include "vga_ball.h"
# include < sys / ioctl.h >
# include < sys / types.h >
# include < sys / stat.h >
# include < fcntl.h >
# include < string.h >
# include < unistd.h >
# include "usbkeyboard.h"
# include "movement.h"
# include "collision.h"
# include < time.h >
```

```
////////////////////define entity//////////////////////////////////////////
# define NO_ID 0
# define SPACESHIP 1
# define BULLET 2
# define SPAWN 3
# define ENEMY4 4
# define ENEMY5 5
# define ENEMY6 6
# define ENEMY7 7
# define EXPLOSION_BULLET 11
# define EXPLOSION_SPACESHIP 12
////////////////////////define direction//////////////////////////////////////
# define LEFT 0
# define LEFTUP 1
# define UP 2
# define RIGHTUP 3
# define RIGHT 4
# define RIGHTDOWN 5
# define DOWN 6
# define LEFTDOWN 7
# define A 22695477
# define B 12983942
# define C 1

struct libusb_device_handle * keyboard;
uint8_t endpoint_address;

int vga_ball_fd;

/* Read and print the segment values */
void print_directions_info() {
	vga_ball_arg_t vla;
	int i;
	for (i = 0; i < 256; i++) {
		vla.axis = i;
		if (ioctl(vga_ball_fd, VGA_BALL_READ_AXIS, & vla)) {
			perror("ioctl(VGA_BALL_READ_AXIS) failed");
			return;
		}
		printf("%02x ", vla.direction);
	}
	printf("\n");
}

/* Write the contents of the array to the display */
void write_directions(const unsigned int directions[256])
{
	vga_ball_arg_t vla;
	int i;
	for (i = 0; i < 256; i++)
	{	vla.axis = i;
		vla.direction = directions[i];
		if (ioctl(vga_ball_fd, VGA_BALL_WRITE_AXIS, & vla)) {
```

```
                  perror("ioctl(VGA_BALL_WRITE_AXIS) failed");
                  return;
            }
      }
}
/*random_number_generation*/
unsigned int random_number_generation_x() {
      unsigned int random_num;
      random_num = rand() % 580 + 1;
      if (random_num <= 32) {
            random_num = 32;
      }
      return random_num;
}
unsigned int random_number_generation_y() {
      unsigned int random_num;
      random_num = rand() % 440 + 1;
      if (random_num <= 32) {
            random_num = 32;
      }
      return random_num;
}

int main() {
      int counter_bullet = 0;
      int n;
      int j, k;
      int flag_bullet = 0;
      int flag_spawn = 0;
      int random_tmp_x = 0;
      int random_tmp_y = 0;
      int spaceship_counter = 0;
      int spawn_counter = 0;
      int counter = 30;
      int movecounter = C;
      int over = 0;
      unsigned int * point_message;
      unsigned int * point_message2;
      unsigned int collision_result;
      unsigned int counter_wait;
      int VGA_audio_collision = 0;
      //signals for player/game info
      int life = 3; //message[240] storing life
      int score = 0; //message[241] storing score
      int bomb = 0; //message[242] storing bomb
      int bomb_status = 0; //used to eliminate button shake infulence
      int game_over = 0; //message[255] storing game_over
      int recounter = 0; //restart game counter
      int h = 0, t = 0, d = 0; //converting scores
      int start = 0; //start game
      int dr = 0; //move  right
      int dd = 0; //move  down
      int temp = 0;
      struct usb_keyboard_packet packet;
      int transferred;
      char keystate[12];
      srand(time(NULL));

      /* Open the keyboard */
      if ((keyboard = openkeyboard( & endpoint_address)) == NULL) {
             fprintf(stderr, "Did not find a keyboard\n");
             exit(1);
      }

      vga_ball_arg_t vla;
      int left, leftup, up, leftdown, down, rightdown, right, rightup;
      int left_d, leftup_d, up_d, leftdown_d, down_d, rightdown_d, right_d, rightup_d;
      int clear_key = 1;

      static const char filename[] = "/dev/vga_ball";
      static unsigned int message[256] = {SPACESHIP, 320, 240, 0 };
      message[240] = 3; //initialize life
      message[242] = 3; //initialize bomb

      //====================================================================
      //end of message initialization
      //====================================================================

      printf("VGA BALL Userspace program started\n");

      if ((vga_ball_fd = open(filename, O_RDWR)) == -1) {
             fprintf(stderr, "could not open %s\n", filename);
             return -1;
      }

      printf("initial state: ");
      print_directions_info();
      write_directions(message);
      printf("current state: ");
      print_directions_info();

      //for loop for starting a new game
      for (;;) {
             //initialize score, life, bomb and message;
             d = 0;
             t = 0;
             h = 0;
             life = 3;
             score = 0;
             bomb = 0;
             bomb_status = 0;
             game_over = 1;
             start = 0;
             static unsigned int message[256] = {SPACESHIP, 320, 240, 0};
             message[240] = 3; //initialize life
```

```
            message[242] = 1; //initialize bomb
            message[244] = 0; //initialize collision audio flag
            message[246] = 0;
            message[247] = 0;
            message[248] = 0;
            message[255] = 1;
            //----------end of initialization-------------

            /* Look for any keypresses from the game controller*/
            for (;;) {
                  if (message[240] == 0) { //sending game_over message
                        for (recounter = 4; recounter < 240; recounter = recounter + 4) {
                              message[recounter] = NO_ID;
                        }
                        message[244] = 0;
                        message[255] = 1;
                        if (packet.keycode[4] == 0x20) //Restart new game
                              break;
                  }
                  if (message[0] == 0) {
                        message[0] = 1;
                  }
                  libusb_interrupt_transfer(keyboard, endpoint_address, (unsigned char * ) & packet,
sizeof(packet), & transferred, 0);
                  if (transferred == sizeof(packet)) {
                        sprintf(keystate, "%02x %02x %02x %02x %02x %02x %02x %02x", packet.modifiers,
packet.reserved, packet.keycode[0],
                              packet.keycode[1], packet.keycode[2], packet.keycode[3],
packet.keycode[4], packet.keycode[5]);
                        printf("%s\n", keystate);
                        if ((packet.modifiers == 0x7f && packet.reserved == 0x7f && packet.keycode[0]
== 0x00 && packet.keycode[1] == 0x80 &&
                              packet.keycode[2] == 0x80 && packet.keycode[3] == 0x0f &&
packet.keycode[4] == 0x00 && packet.keycode[5] == 0x00) || clear_key == 1) {
                              left = 0;
                              leftup = 0;
                              up = 0;
                              leftdown = 0;
                              down = 0;
                              rightdown = 0;
                              right = 0;
                              rightup = 0;
                              left_d = 0;
                              leftup_d = 0;
                              up_d = 0;
                              leftdown_d = 0;
                              down_d = 0;
                              rightdown_d = 0;
                              right_d = 0;
                              rightup_d = 0;
                              bomb = 0;
                        }
                        /*BOMB BUTTON DETECTION*/
```

```
                            if (packet.keycode[4] == 0x08 && message[242] != 0 && bomb == 0 && bomb_status
< 30) {
                                  bomb_status = bomb_status + 1;
                                  if (bomb_status == 30) {
                                        bomb_status = 0;
                                        bomb = 1;
                                        message[242] = message[242] - 1;
                                  }
                            }
                            /*Start a new game*/
                            if (packet.keycode[4] == 0x20) {
                                  start = 1;
                                  game_over = 0;
                                  message[255] = 0;
                            }
                            if (message[255] == 0 && message[240] > 0) {

                                  if (packet.keycode[3] == 0x4f) {
                                        down_d = 1;
                                  }
                                  if (packet.keycode[3] == 0xcf) {
                                        leftdown_d = 1;
                                  }
                                  if (packet.keycode[3] == 0x8f) {
                                        left_d = 1;
                                  }
                                  if (packet.keycode[3] == 0x9f) {
                                        leftup_d = 1;
                                  }
                                  if (packet.keycode[3] == 0x1f) {
                                        up_d = 1;
                                  }
                                  if (packet.keycode[3] == 0x3f) {
                                        rightup_d = 1;
                                  }
                                  if (packet.keycode[3] == 0x2f) {
                                        right_d = 1;
                                  }
                                  if (packet.keycode[3] == 0x6f) {
                                        rightdown_d = 1;
                                  }

                                  if (packet.modifiers == 0x00) {
                                        if (packet.reserved == 0x7f) {
                                              left = 1;
                                        }
                                        if (packet.reserved == 0x00) {
                                              leftup = 1;
                                        }
                                        if (packet.reserved == 0xff) {
                                              leftdown = 1;
                                        }
                                  }
```

```
                    if (packet.modifiers == 0xff) {
                          if (packet.reserved == 0x7f) {
                                right = 1;
                          }
                          if (packet.reserved == 0x00) {
                                rightup = 1;
                          }
                          if (packet.reserved == 0xff) {
                                rightdown = 1;
                          }
                    }
                    if (packet.modifiers == 0x7f) {
                          if (packet.reserved == 0x00) {
                                up = 1;
                          }
                          if (packet.reserved == 0xff) {
                                down = 1;
                          }
                    }
              }
              write_directions(message);
              if (!game_over && start) {
                    if (right == 1) {
                          if (message[1] < 580) //640-64 message[1] is x coordinate of
spaceship,

                          {
                                message[1]++;
                          }
                    }
                    if (left == 1) {
                          if (message[1] != 32) //leftup
                          {
                                message[1]--;
                          }
                    }
                    if (down == 1) {
                          if (message[2] < 428) //480-64
                          {
                                message[2]++;
                          }
                    }
                    if (up == 1) {
                          if (message[2] != 32) //message[2] is the y coordinate
                          {
                                message[2] = message[2] - 1;
                          }
                    }
                    if (leftup == 1) {
                          if (message[2] != 32) {
                                message[2] = message[2] - 1;
                          }
                          if (message[1] != 32) {
                                message[1]--;
                          }
```

```
                                usleep(7500);
                        }
                        if (leftdown == 1) {
                                if (message[2] < 428) {
                                        message[2]++;
                                }
                                if (message[1] != 32) {
                                        message[1]--;
                                }
                                usleep(7500);
                        }
                        if (rightdown == 1) {
                                if (message[2] < 428) {
                                        message[2]++;
                                }
                                if (message[1] < 580) {
                                        message[1]++;
                                }
                                usleep(7500);
                        }
                        if (rightup == 1) {
                                if (message[2] != 32) {
                                        message[2] = message[2] - 1;
                                }
                                if (message[1] < 580) {
                                        message[1]++;
                                }

                                usleep(7500);
                        }
                        //======================================================================
                        //end of avatar position control
        //======================================================================
                        //bullet data generation
                        /* add counter to control the speed to enter the following for loop */
                        if (counter_bullet == 15) {
                                int i;
                                for (i = 4; i < 120; i = i + 4) //The bullet info start from
message[4]
                                {
                                        //generate the id of the bullet
                                        if (message[i] == 0) {
                                                flag_bullet = 1;
                                                message[i] = BULLET; //define the id
                                                if (left_d == 1) {
                                                        message[i + 3] = LEFT;
                                                }
                                                if (leftup_d == 1) {
                                                        message[i + 3] = LEFTUP;
                                                }
                                                if (leftdown_d == 1) {
                                                        message[i + 3] = LEFTDOWN;
                                                        usleep(7500);
                                                }
                        if (down_d == 1) {
                            message[i + 3] = DOWN;
                        }
                        if (right_d == 1) {
                            message[i + 3] = RIGHT;
                        }
                        if (rightdown_d == 1) {
                            message[i + 3] = RIGHTDOWN;
                            usleep(7500);
                        }
                        if (rightup_d == 1) {
                            message[i + 3] = RIGHTUP;
                            usleep(7500);
                        }
                        if (up_d == 1) {
                            message[i + 3] = UP;
                        }
                        switch (message[i + 3]) {
                            case LEFT:
                                {
                                    message[i + 1] = message[1] - 18;
                                    message[i + 2] = message[2] - 3;
//bullet calibration [8]

                                    break;
                                }
                            case LEFTUP:
                                {
                                    message[i + 1] = message[1] - 10;
                                    message[i + 2] = message[2] - 10;
                                    break;
                                }
                            case UP:
                                {
                                    message[i + 1] = message[1] + 3;
                                    message[i + 2] = message[2] - 16;
                                    break;
                                }
                            case RIGHTUP:
                                {
                                    message[i + 1] = message[1] + 15;
                                    message[i + 2] = message[2] - 10;
                                    break;
                                }
                            case RIGHT:
                                {
                                    message[i + 1] = message[1] + 18;
                                    message[i + 2] = message[2];
                                    break;
                                }
                            case RIGHTDOWN:
                                {
                                    message[i + 1] = message[1] + 10;
                                    message[i + 2] = message[2] + 10;
                                    break;
```

```
                                                                    }
                                                             case DOWN:
                                                                    {
                                                                           message[i + 1] = message[1];
                                                                           message[i + 2] = message[2] + 16;
                                                                           break;
                                                                    }
                                                             case LEFTDOWN:
                                                                    {
                                                                           message[i + 1] = message[1] - 10;
                                                                           message[i + 2] = message[2] + 10;
                                                                           break;
                                                                    }
                                                             default:
                                                                    {
                                                                           message[i + 1] = message[1] - 16;
                                                                           message[i + 2] = message[2];
                                                                           break;
                                                                    }
                                                      }
                                               }
                                               if (flag_bullet) {
                                                      flag_bullet = 0;
                                                      break;
                                               }

                                        }
                                        counter_bullet = 0;

                                 } else {
                                        counter_bullet++;
                                 }
                                 //===================================================================
                   //end of bullet data generation control
      //===================================================================
                                 //===================================================================
                   //enter spawn entity/enemy generation
      //===================================================================

                                 counter_wait = random_number_generation_x();
                                 //control the speed of enermy generation
                                 if (counter_wait % 23 == 0) {
                                        int q, z, spawn_counter = 0;;
                                        for (q = 120; q < 240; q = q + 4) //address start from the 30th, i
start from 30*4
                                        {
                                               if (message[q] == NO_ID) {
                                                      message[q + 1] = random_number_generation_x();
                                                      message[q + 2] = random_number_generation_y();
                                                      for (z = 120; z < 240; z = z + 4) {
                                                             if ((abs(message[q + 1] - message[z + 1]) >= 32
|| abs(message[q + 2] - message[z + 2]) >= 32) && (abs(message[q] - message[1] >= 32))) //-----32
                                                             {
                                                                    spawn_counter++;
                                                             }
                                                      }
                                                      if (spawn_counter == 29) {
                                                             message[q] = 3;
                                                             flag_spawn = 1;
                                                             spawn_counter = 0;
                                                      } else {
                                                             message[q] = 0;
                                                             message[q + 1] = 0;
                                                             message[q + 2] = 0;
                                                             flag_spawn = 1;
                                                             spawn_counter = 0;
                                                      }
                                                      message[q + 3] = 0;
                                               }
                                               if (flag_spawn) {
                                                      flag_spawn = 0;
                                                      break;
                                               }
                                        }
                                 }
                                 //===================================================================
                                 //end of generate spawn/enemy data generation and movement control
                           //===================================================================
                                 //===================================================================
                                 //enter movement decision
      //===================================================================
                                 for (n = 4; n < 240; n = n + 4) //the last object start from message[236]
                                 {
                                        int a = 1, b = 1;
                                        int x_old, y_old;
                                        x_old = message[n + 1];
                                        y_old = message[n + 2];
                                        if (message[n] == NO_ID) {
                                               message[n + 1] = 0;
                                               message[n + 2] = 0;
                                               message[n + 3] = 0;
                                        }
                                        if (message[n] == BULLET) {
                                               if (message[n + 1] > 580 || message[n + 2] > 440 ||
message[n + 1] < 32 || message[n + 2] < 32) {
                                                      message[n] = NO_ID;
                                               } else if (message[n + 3] == LEFT) {
                                                      message[n] = BULLET;
                                                      message[n + 1] = message[n + 1] - 4;
                                                      message[n + 2] = message[n + 2];
                                                      message[n + 3] = message[n + 3];
                                               } else if (message[n + 3] == LEFTUP) {
                                                      message[n] = BULLET;
                                                      message[n + 1] = message[n + 1] - 4;
                                                      message[n + 2] = message[n + 2] - 4;
                                                      message[n + 3] = message[n + 3];
                                               } else if (message[n + 3] == LEFTDOWN) {
```

```
                                    message[n] = BULLET;
                                    message[n + 1] = message[n + 1] - 4;
                                    message[n + 2] = message[n + 2] + 4;
                                    message[n + 3] = message[n + 3];
                              } else if (message[n + 3] == DOWN) {
                                    message[n] = BULLET;
                                    message[n + 1] = message[n + 1];
                                    message[n + 2] = message[n + 2] + 4;
                                    message[n + 3] = message[n + 3];
                              } else if (message[n + 3] == RIGHT) {
                                    message[n] = BULLET;
                                    message[n + 1] = message[n + 1] + 4;
                                    message[n + 2] = message[n + 2];
                                    message[n + 3] = message[n + 3];
                              } else if (message[n + 3] == RIGHTDOWN) {
                                    message[n] = BULLET;
                                    message[n + 1] = message[n + 1] + 4;
                                    message[n + 2] = message[n + 2] + 4;
                                    message[n + 3] = message[n + 3];
                              } else if (message[n + 3] == RIGHTUP) {
                                    message[n] = BULLET;
                                    message[n + 1] = message[n + 1] + 4;
                                    message[n + 2] = message[n + 2] - 4;
                                    message[n + 3] = message[n + 3];
                              } else if (message[n + 3] == UP) {
                                    message[n] = BULLET;
                                    message[n + 1] = message[n + 1];
                                    message[n + 2] = message[n + 2] - 4;
                                    message[n + 3] = message[n + 3];
                              }
                        } else if (message[n] == SPAWN) {
                              if (message[n + 1] > 580 || message[n + 2] > 440 ||
message[n + 1] < 32 || message[n + 2] < 32) // when out of board, turn into no_id immediately
                              {
                                    message[n] = NO_ID;
                              }
                              if (counter == 30) {

                                    if (message[n + 3] == 0) {

                                          message[n] = SPAWN;
                                          message[n + 1] = message[n + 1];
                                          message[n + 2] = message[n + 2];
                                          message[n + 3] = 1;
                                    } else if (message[n + 3] == 1) {
                                          message[n] = SPAWN;
                                          message[n + 1] = message[n + 1];
                                          message[n + 2] = message[n + 2];
                                          message[n + 3] = 2;
                                    } else if (message[n + 3] == 2) {
                                          message[n] = SPAWN;
                                          message[n + 1] = message[n + 1];
                                          message[n + 2] = message[n + 2];
```

```
                                        message[n + 3] = 3;
                                } else if (message[n + 3] == 3) {
                                        switch (message[n + 1] % 4) {
                                                case 0:
                                                        message[n] = ENEMY4;
                                                        break;
                                                case 1:
                                                        message[n] = ENEMY5;
                                                        break;
                                                case 2:
                                                        message[n] = ENEMY6;
                                                        break;
                                                case 3:
                                                        message[n] = ENEMY7;
                                                        break;
                                                default:
                                                        message[n] = ENEMY6;
                                                        break;
                                        }
                                        message[n + 1] = message[n + 1];
                                        message[n + 2] = message[n + 2];
                                        message[n + 3] = 0;
                                }
                                counter = 0;
                        } else counter++;
                }

                //SPAWN TURN INTO ENEMY4, bouncing ---bluecirle
                else if (message[n] == ENEMY4) {
                        if (message[n + 1] > 580 || message[n + 2] > 440 ||
message[n + 1] < 32 || message[n + 2] < 32) {
                                message[n] = NO_ID;
                        } else {
                                message[n] = ENEMY4;
                                if (dr) message[n + 1] = message[n + 1] + a;
                                else message[n + 1] = message[n + 1] - a;
                                if (dd) message[n + 2] = message[n + 2] + a;
                                else message[n + 2] = message[n + 2] - a;
                                if (message[n + 1] >= 580) dr = 0;
                                else if (message[n + 1] <= 64) dr = 1;
                                if (message[n + 2] >= 440) dd = 0;
                                else if (message[n + 2] <= 64) dd = 1;

                        }
                        message[n + 3] = 0;
                }

                //SPAWN TURN INTO ENEMY5, straight --- redcircle
                else if (message[n] == ENEMY5) {
                        if (message[n + 1] > 580 || message[n + 2] > 440 ||
message[n + 1] < 32 || message[n + 2] < 32) {
                                message[n] = NO_ID;
                        } else {
```

```
                                                message[n] = ENEMY5;
                                                message[n + 1] = message[n + 1] - a;
                                                message[n + 2] = message[n + 2] + a;
                                        }
                                        message[n + 3] = 0;
                                }

                                //SPAWN  TURN  INTO ENEMY6, chase after the spaceship --triangle
                                else if (message[n] == ENEMY6) {
                                        if (message[n + 1] > 580 || message[n + 2] > 440 ||
message[n + 1] < 32 || message[n + 2] < 32) {
                                                message[n] = NO_ID;
                                        }
                                        if (1) {
                                                message[n] = ENEMY6;
                                                if (message[1] >= message[n + 1]) {
                                                        message[n + 1] = message[n + 1] + 1;
                                                } else {
                                                        message[n + 1] = message[n + 1] - 1;
                                                }
                                                if (message[2] >= message[n + 2]) {
                                                        message[n + 2] = message[n + 2] + 1;
                                                } else {
                                                        message[n + 2] = message[n + 2] - 1;
                                                }
                                                movecounter = 0;
                                        }
                                        message[n + 3] = 0;
                                }

                                //SPAWN  TURN  INTO ENEMY7, chase----rectangular
                                else if (message[n] == ENEMY7) {
                                        if (message[n + 1] > 580 || message[n + 2] > 440 ||
message[n + 1] < 32 || message[n + 2] < 32) {
                                                message[n] = NO_ID;
                                        }
                                        if (1) {
                                                message[n] = ENEMY7;
                                                if (message[1] >= message[n + 1]) {
                                                        message[n + 1] = message[n + 1] - 1;
                                                } else {
                                                        message[n + 1] = message[n + 1] + 1;
                                                }
                                                if (message[2] >= message[n + 2]) {
                                                        message[n + 2] = message[n + 2] - 1;
                                                } else {
                                                        message[n + 2] = message[n + 2] + 1;
                                                }
                                                movecounter = 0;
                                        }
                                        message[n + 3] = 0;
                                }
                                else if (message[n] == EXPLOSION_BULLET) {
```

```
                                                if (message[n + 1] > 580 || message[n + 2] > 440 ||
message[n + 1] < 32 || message[n + 2] < 32) {
                                                        message[n] = NO_ID;
                                                }
                                                if (counter == 30) {
                                                        if (message[n + 3] == 0) {
                                                                message[n] = EXPLOSION_BULLET;
                                                                message[n + 1] = message[n + 1];
                                                                message[n + 2] = message[n + 2];
                                                                message[n + 3] = 1;
                                                        } else if (message[n + 3] == 1) {
                                                                message[n] = NO_ID;
                                                                message[n + 1] = 0;
                                                                message[n + 2] = 0;
                                                                message[n + 3] = 0;
                                                        }
                                                        counter = 0;
                                                } else counter++;
                                        } else if (message[n] == NO_ID) {
                                                message[n] = 0;
                                                message[n + 1] = 0;
                                                message[n + 2] = 0;
                                                message[n + 3] = 0;
                                        }
                                }

                                //=====================================================================
                                //detect collision and added bombs
                                //=====================================================================
                                int bomb_clear;
                                for (j = 0; j < 120; j = j + 4) {
                                        for (k = 120; k < 240; k = k + 4) {
                                                if ((message[242] >= 0 && bomb == 1) || bomb_clear == 1) {
                                                        message[k] = EXPLOSION_BULLET;
                                                        bomb = 0;
                                                        bomb_clear = 1;
                                                        if (k == 236)
                                                                bomb_clear = 0;
                                                } else {
                                                        collision_result = collision(message[j], message[j +
1], message[j + 2], message[k], message[k + 1], message[k + 2]);
                                                        if (collision_result == 1) { // a collision happens
                                                                VGA_audio_collision = 1;
                                                                message[244] = VGA_audio_collision;
                                                                if (message[j] == SPACESHIP) {
                                                                        life = life - 1;
                                                                        if (life == 0) {
                                                                                game_over = 1;
                                                                                message[244] = 0;
                                                                        }
                                                                        for (over = 0; over < 240; over = over +
4)
                                                                                message[over] = NO_ID;
```

```
                                                            } else if (message[k] == ENEMY4 || message[k]
== ENEMY5 || message[k] == ENEMY6 || message[k] == ENEMY7 || message[k] == SPAWN) {
                                                                   message[j] = NO_ID;
                                                                   message[k] = EXPLOSION_BULLET;
                                                                   message[k + 3] = 0;
                                                                   score = score + 1;
                                                            } else if (message[k] == EXPLOSION_BULLET &&
message[k + 3] == 1) {
                                                                   message[j] = NO_ID;
                                                                   message[k] = NO_ID;
                                                            } else if (message[k] == EXPLOSION_BULLET &&
message[k + 3] == 0) {
                                                                   message[j] = NO_ID;
                                                                   message[k] = EXPLOSION_BULLET;
                                                            }
                                                     }
                                              }
                                       }
                                }
                                if (VGA_audio_collision > 0) {
                                       message[244] = 1;
                                       VGA_audio_collision = 0;
                                } else message[244] = 0;
                                //=====================================================================
                                //end of detect collision
                                //=====================================================================

                                //sending player informaiton
                                //score conversion
                                d = score % 10;
                                t = ((score - d) / 10) % 10;
                                h = ((score - 10 * t - d) / 100) % 10;
                                message[246] = h; //hundreds
                                message[247] = t; //tens
                                message[248] = d; //digits
                                message[240] = life;
                                message[241] = score;
                                message[255] = game_over;
                         }
                  }
           }
       }
       printf("VGA LED Userspace program terminating\n");
       return 0;
}
```

5.VGA_BALL_Emulator.sv

```
 /*
 * VGA_BALL_Emulator
 * receive info from VGA_BALL.sv
 * use the info to draw sprites
 * by using line buffers to solving
 * overlap issurs.
 *
 * Zhenyu Zhu, Columbia University
 */

module VGA_BALL_Emulator(
  input logic clk50, reset,
  input logic enable_ram, //enable ram to receieve data from VGA_ball.sv
  input logic ram_select, //select which ram is used to store data
       input logic [1:0] life,
       input logic [1:0] bomb_value,
       input logic [9:0] score,
       input logic [3:0] tens,
       input logic [3:0] digits,
       input logic game_over,
       input logic en_emulator,//tell ram to store data
  input logic update_done,//VGA_ball.sv says you can work
  input logic [31:0] game_logic_to_emulator, //object info is sent by this
       output logic draw_done,//tell game logic you finish
       output logic [7:0] VGA_R, VGA_G, VGA_B,
       output [31:0]A,B,C,
       output [31:0]mem_out_u,mem_out,A_in,B_in,C_in,ram_out,
       output logic  VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);


  // Horizontal counter
  logic [10:0]    hcount;
  logic           endOfLine;

  // Vertical counter
  logic [9:0]     vcount;
  logic           endOfField;


  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

  //ID for sprites
  parameter NO_ID          =8'd0,
                      SPACESHIP=8'd1,
              BULLET            =8'd2,
                      SPAWN          =8'd3,
                      ENEMY4         =8'd4,
                      ENEMY5         =8'd5,
                      ENEMY6         =8'd6,
                      ENEMY7         =8'd7,
                     EXPLOSION_BULLET = 8'd11,
                      EXPLOSION_SPACESHIP = 8'd12;

  //aixes for current pixels on the screen
  logic  [9:0]    xcoor;
  logic  [9:0]    ycoor;

  //sprites color info
  logic [23:0] color_spaceship;
  logic [23:0] color_background;
  logic [23:0] color_bullet;
  logic [23:0] color_spawn0;
  logic [23:0] color_spawn45;
  logic [23:0] color_enemy400;
  logic [23:0] color_enemy500;
  logic [23:0] color_enemy600;
  logic [23:0] color_enemy700;
  logic [23:0] color_explosion_bullet;
  logic [23:0] color_explosion_bullet180;
  logic [23:0] color_explosion_spaceship;
  logic [23:0] color_explosion_spaceship1;
  logic [23:0] color_lifeword;
  logic [23:0] color_number_zero;
  logic [23:0] color_number_one;
  logic [23:0] color_number_two;
  logic [23:0] color_number_three;
  logic [23:0] color_number_four;
  logic [23:0] color_number_five;
  logic [23:0] color_number_six;
  logic [23:0] color_number_seven;
  logic [23:0] color_number_eight;
  logic [23:0] color_number_nine;
  logic [23:0] color_bomb;
  logic [23:0] color_score;
  logic [23:0] color_start;
  logic [23:0] color_starwars;

  //sprite address
  logic  [9:0]   space_address;
  logic  [11:0]  background_address;
  logic  [9:0]   bullet_rom_address;
  logic  [9:0]   spawn_address;
  logic  [9:0]   enemy4_address;
  logic  [9:0]   enemy5_address;
  logic  [9:0]   enemy6_address;
  logic  [9:0]   enemy7_address;
  logic  [9:0]   explosion_bullet_address;
  logic  [9:0]   explosion_spaceship_address;
  logic  [11:0]  lifeword_address;
  logic  [9:0]   number_address;
  logic  [12:0]  bombword_address;
  logic  [11:0]  scoreword_address;
  logic  [14:0]  startword_address;
  logic  [14:0]  startwarsword_address;
```

```
  //states for the total control of the emulator
  enum logic [2:0] {waiting, processing} states;
  //states for receive data from outside
  enum logic [1:0] {wait_receive_finish,receive_data} states_process;
  //states for line buffers to work
  enum logic [3:0] {line_buffer_setting, line_buffer_ram,line_buffer_waiting, line_buffer_working}
line_buffer_states;
  //states for continuosly writing data in line buffers
  enum logic [4:0] {loop_waiting, loop_tmp_waiting, loop_tmp_setting,loop_writing} loop_states;
  //states for drawing graph
  enum logic [1:0] {draw_waiting, draw_processing} draw_states;
  //states for choosing line buffers
  enum logic [2:0] {uA_cB_dC, uB_cC_dA,uC_cA_dB} line_buffer_select;

  //This always blcok Used to control the drawing process.
  always_ff @(posedge clk50 or posedge reset) begin
      if (reset) begin
          hcount <= 0;
          vcount <= 0;
          states <= waiting;
          draw_done <= 0;
      end
      else if (states == waiting) begin
          hcount <= 0;
          vcount <= 0;
          draw_done <= 0;
                 //states_process <= receive_data;
          if (update_done == 1'd1) begin
              states <= processing;
          end
      end
      else if (states == processing) begin
          if(endOfLine) begin
              hcount <= 0;
              if (endOfField) begin
                  vcount <= 0;
                  draw_done <= 1;
                  states <= waiting;
              end
              else begin
                  vcount <= vcount + 10'd1;
              end
          end
          else begin
              hcount <= hcount +11'd1;
          end
      end
  end

  assign endOfLine = hcount == HTOTAL - 1;
  assign endOfField = vcount == VTOTAL - 1;

  /*
  * 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          |____|
 */


 // 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)
);

 //info about rams to store incoming data
 logic [7:0] ram_addr;
      logic [7:0] ram_counter;
      logic ram_a_we,ram_b_we;
      logic ram_we;
      logic [31:0] ram_a_in,ram_b_in,ram_in;
      logic [31:0] ram_a_out,ram_b_out;
      logic [7:0] ram_a_address,ram_b_address;
 assign ram_a_we = ram_select ? ram_we:0;//ram_selct = 1, ram_a receives data from vga_BALL.
 assign ram_b_we = ram_select ? 0:ram_we; //ram_selct = 0, ram_b receives data from vga_BALL.
 assign ram_a_in = ram_select ? ram_in:0;//ram_selct = 1, ram_a receives data from vga_BALL.
 assign ram_b_in = ram_select ? 0:ram_in;//ram_selct = 0, ram_b receives data from vga_BALL.
 assign ram_a_address = ram_select ? ram_addr:ram_counter;//ram_selct = 1, ram_a receives data from
vga_BALL.
 assign ram_b_address = ram_select ? ram_counter:ram_addr;//ram_selct = 1, used to draw graph
 assign ram_out = ram_select ? ram_b_out:ram_a_out;//ram_selct = 0, ram_a used to draw graph

 //instantiation for rams
 myram
ram_a_tmp(.clk(clk50),.we(ram_a_we),.data_in(ram_a_in),.data_out(ram_a_out),.address(ram_a_address));
 myram
ram_b_tmp(.clk(clk50),.we(ram_b_we),.data_in(ram_b_in),.data_out(ram_b_out),.address(ram_b_address));

 //This block used to receive incoming data from VGA_ball.sv
 always_ff @(posedge clk50) begin
   if (states == waiting) begin
             states_process <= receive_data;
             ram_addr <= 0;
         end
   else if(enable_ram == 1'd1) begin //start to update data from ram
```

```
            if (states_process == receive_data) begin
                  if (en_emulator) begin
                        ram_in <= game_logic_to_emulator;
                        states_process <= wait_receive_finish;
                        ram_we <= 1;
                  end
            end
            else if (states_process == wait_receive_finish) begin
                        ram_addr <= ram_addr + 8'd1;
                        ram_we <= 0;
                        states_process <= receive_data;
            end
    end
    else begin
          ram_addr <= 8'd0; //next update start from ram_addr = 0;
        end
  end

  //information about the line buffer
  logic we;
  logic A_we;
  logic B_we;
  logic C_we;
  logic update_we;
  logic clean_we;
  logic [9:0] buffer_A_address;
  logic [9:0] buffer_B_address;
  logic [9:0] buffer_C_address;
  logic [9:0] buffer_update_address;
  logic [9:0] buffer_clean_address;
  logic [9:0] buffer_draw_address;
  logic [31:0]mem_in_update;
  logic [31:0]mem_in_clean;
  logic [31:0]mem_in_A;
  logic [31:0]mem_in_B;
  logic [31:0]mem_in_C;
  logic [31:0]A_out;
  logic [31:0]B_out;
  logic [31:0]C_out;
  assign A = A_out;
  assign B = B_out;
  assign C = C_out;
  assign A_in = mem_in_A;
  assign B_in = mem_in_B;
  assign C_in = mem_in_C;

  //instantiation for line buffers
  line_buffer
line_buffer_A(.clk(clk50),.we(A_we),.data_in(mem_in_A),.data_out(A_out),.address(buffer_A_address));
  line_buffer
line_buffer_B(.clk(clk50),.we(B_we),.data_in(mem_in_B),.data_out(B_out),.address(buffer_B_address));
  line_buffer
line_buffer_C(.clk(clk50),.we(C_we),.data_in(mem_in_C),.data_out(C_out),.address(buffer_C_address));
```

```
//This always block used to select which line buffer is used to recevie
//the newest data, which is used to draw graph, which is used prepare
//for rececing data next time.
always_comb begin
           A_we = 0;
  B_we = 0;
           C_we = 0;
  mem_in_A = 32'd0;
  mem_in_B = 32'd0;
           mem_in_C = 32'd0;
  buffer_A_address = 9'd0;
           buffer_B_address = 9'd0;
           buffer_C_address = 9'd0;
  if (line_buffer_select == uA_cB_dC) begin
    A_we = update_we;
    B_we = clean_we;
    mem_in_A = mem_in_update ;
    mem_in_B = mem_in_clean;
    buffer_A_address = buffer_update_address;
    buffer_B_address = buffer_clean_address;
             buffer_C_address = buffer_draw_address;
  end
  else if (line_buffer_select == uB_cC_dA) begin
    B_we = update_we;
    C_we = clean_we;
    mem_in_B = mem_in_update ;
    mem_in_C = mem_in_clean;
    buffer_B_address = buffer_update_address;
    buffer_C_address = buffer_clean_address;
             buffer_A_address = buffer_draw_address;
  end
  else  if (line_buffer_select == uC_cA_dB) begin
    C_we = update_we;
    A_we = clean_we;
    mem_in_C = mem_in_update ;
    mem_in_A = mem_in_clean;
    buffer_C_address = buffer_update_address;
    buffer_A_address = buffer_clean_address;
             buffer_B_address = buffer_draw_address;
  end
end

assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on rising edge
assign xcoor = hcount[10:1]; //link hcount and location of current pixels
assign ycoor = vcount; //link vcount
logic [4:0] store_counter; //help choose consecutive location in line buffers
logic jump_flag1; //help to do state transfer
logic [31:0] object_info;
assign object_info = ram_out;//link the ram out to object info

//This always block defines how three lines work
//The first part is used to update the line buffer
//The second part is used to draw graphs and clean
//line buffer
```

```
  always_ff @(posedge clk50) begin
    if (reset) begin
                line_buffer_states <= line_buffer_setting;
                draw_states <= draw_waiting;
                     loop_states <= loop_writing;
                     store_counter <= 0;
                     line_buffer_select <= uA_cB_dC;
                     jump_flag1 <= 0;
                     space_address <= 10'd0;
                     bullet_rom_address <= 10'd0;
                     spawn_address <= 10'd0;
         end
    //update one of three lines for next drawing
    else if (states == processing) begin
      if(line_buffer_states == line_buffer_setting) begin
        line_buffer_states <= line_buffer_working;
      end
             if (line_buffer_states == line_buffer_working) begin
        //check is there anything in this line defined by vcount
        if (vcount - object_info[13:4] >= 0 && vcount - object_info[13:4] <= 10'd31 ) begin
          //determine positions in line buffers to store RGB info
          if (loop_states == loop_writing) begin
            update_we <= 0;
            //if fisrt pixel of the line in a image, use current x aixs, indicate where the data at the
line buffer.
            if (store_counter == 5'd0) begin
              buffer_update_address <= object_info[23:14];
              if (jump_flag1 != 1) loop_states <= loop_tmp_waiting;
            end
            //if the lastpixel of the line in a image, finish, go to fetch next value in ram
            else if(store_counter == 5'd31) begin
              loop_states <= loop_writing;
              jump_flag1 <= 1;
              store_counter <= 0;
            end
            else begin
                                 buffer_update_address <= buffer_update_address + 1;
                            loop_states <= loop_tmp_waiting;
                       end
           end
                     else if(loop_states == loop_tmp_waiting)begin
                          loop_states <= loop_waiting;
            //determing the RGB value
                          if (object_info[31:24] == SPACESHIP) begin
                               space_address <= (buffer_update_address-object_info[23:14])+(vcount-
object_info[13:4])*32;
                               if (color_spaceship != 24'd0) update_we <= 1;
                                mem_in_update <= {object_info[31:24],color_spaceship[23:16],
color_spaceship[15:8], color_spaceship[7:0]};
                          end
                          else if (object_info[31:24] == BULLET) begin
                               bullet_rom_address <= (buffer_update_address-object_info[23:14])+(vcount-
object_info[13:4])*32;
                               if (color_bullet != 24'd0) update_we <= 1;
```

```
                                mem_in_update <= {object_info[31:24],color_bullet[23:16],
color_bullet[15:8], color_bullet[7:0]};
                          end
                          else if (object_info[31:24] == SPAWN) begin
                                spawn_address <= (buffer_update_address-object_info[23:14])+(vcount-
object_info[13:4])*32;
                                if (object_info[3:0] == 4'd0 || object_info[3:0] == 4'd2) begin
                                  if (color_spawn0 != 24'd0) update_we <= 1;
                                  mem_in_update <= {object_info[31:24],color_spawn0[23:16],
color_spawn0[15:8], color_spawn0[7:0]};
                                 end
                                 else if (object_info[3:0] == 4'd1 || object_info[3:0] == 4'd3) begin
                                        if (color_spawn45 != 24'd0) update_we <= 1;
                                        mem_in_update <= {object_info[31:24],color_spawn45[23:16],
color_spawn45[15:8], color_spawn45[7:0]};
                                  end
                            end
                          else if (object_info[31:24] ==ENEMY4) begin
                                 enemy4_address <= (buffer_update_address-object_info[23:14])+(vcount-
object_info[13:4])*32;
                                 if (color_enemy400 != 24'd0) update_we <= 1;
                                 mem_in_update <= {object_info[31:24],color_enemy400[23:16],
color_enemy400[15:8], color_enemy400[7:0]};
                          end
                          else if (object_info[31:24] ==ENEMY5) begin
                                 enemy5_address <= (buffer_update_address-object_info[23:14])+(vcount-
object_info[13:4])*32;
                                 if (color_enemy500 != 24'd0) update_we <= 1;
                                 mem_in_update <= {object_info[31:24],color_enemy500[23:16],
color_enemy500[15:8], color_enemy500[7:0]};
                          end
                          else if (object_info[31:24] ==ENEMY6) begin
                                 enemy6_address <= (buffer_update_address-object_info[23:14])+(vcount-
object_info[13:4])*32;
                                 if (color_enemy600 != 24'd0) update_we <= 1;
                                 mem_in_update <= {object_info[31:24],color_enemy600[23:16],
color_enemy600[15:8], color_enemy600[7:0]};
                          end
                          else if (object_info[31:24] ==ENEMY7) begin
                                 enemy7_address <= (buffer_update_address-object_info[23:14])+(vcount-
object_info[13:4])*32;
                                 if (color_enemy700 != 24'd0) update_we <= 1;
                                 mem_in_update <= {object_info[31:24],color_enemy700[23:16],
color_enemy700[15:8], color_enemy700[7:0]};
                          end
                          else if (object_info[31:24] ==EXPLOSION_BULLET) begin
                                 explosion_bullet_address <= (buffer_update_address-
object_info[23:14])+(vcount-object_info[13:4])*32;
                                 if (object_info[3:0] == 4'b0000) begin
                                        if (color_explosion_bullet != 24'd0) update_we <= 1;
                                        mem_in_update <=
{object_info[31:24],color_explosion_bullet[23:16], color_explosion_bullet[15:8],
color_explosion_bullet[7:0]};
                                 end
                                else if (object_info[3:0] == 4'b0001) begin
                                      if (color_explosion_bullet180 != 24'd0) update_we <= 1;
                                      mem_in_update <=
{object_info[31:24],color_explosion_bullet180[23:16], color_explosion_bullet180[15:8],
color_explosion_bullet180[7:0]};
                                end
                   end
         //preapre for next pixel in buffer line
         else if(loop_states == loop_waiting) begin
                update_we <= 0;
           store_counter <= store_counter + 1;
           loop_states <= loop_writing;
         end
       end
       else jump_flag1 <= 1;
       if(jump_flag1) begin
         line_buffer_states <= line_buffer_waiting;
         jump_flag1 <=0;
       end
     end
     //choose a new value from ram, if no more value fetched
     //wait for endOfLine and restart from another line buffer
     if (line_buffer_states == line_buffer_waiting) begin
       if(ram_counter == 8'd59) begin
          if (endOfLine) begin
                    ram_counter <=0;
                    if(line_buffer_select == uA_cB_dC) begin
                         buffer_update_address <= 10'd0;
                         buffer_clean_address <= 10'd0;
                         line_buffer_select <= uB_cC_dA;
                    end
                    else if(line_buffer_select == uB_cC_dA) begin
                         buffer_update_address <= 10'd0;
                               buffer_clean_address <= 10'd0;
                         line_buffer_select <= uC_cA_dB;
                    end
                    else if (line_buffer_select == uC_cA_dB) begin
                         buffer_update_address <= 10'd0;
                               buffer_clean_address <= 10'd0;
                         line_buffer_select <= uA_cB_dC;
                    end
                   line_buffer_states <= line_buffer_setting;
                   clean_we <= 0;
            end
            else line_buffer_states <= line_buffer_waiting;
       end
       else begin
                   ram_counter <= ram_counter + 1;
                   line_buffer_states <= line_buffer_setting;
            end
         update_we <= 0;
     end
   end
            //drawing,use two lines buffers, one is used
```

```
    //to draw, the other is used to prepare for
    //next update.
             if (draw_states == draw_waiting) begin
               buffer_draw_address <= xcoor;
                    buffer_clean_address <= xcoor;
                    draw_states <= draw_processing;
                    clean_we <= 1;
                    mem_in_clean <= 32'd0;
             end
             if (draw_states == draw_processing) begin
        //draw life word
                if (xcoor >= 10'd0 && (xcoor <= 10'd96) && (vcount >= 0) && (vcount <= 10'd29)) begin
                          lifeword_address <= (xcoor)+(vcount)*97;
                          {VGA_R, VGA_G, VGA_B} <= {color_lifeword[23:16], color_lifeword[15:8],
color_lifeword[7:0]};
                    end
      //draw life number
                    else if (xcoor >= 10'd97 && (xcoor <= 10'd97+10'd31) && (vcount >= 0) && (vcount <=
10'd31)) begin
                          number_address <= (xcoor-10'd97)+(vcount)*32;
                          if (life == 3'd0) {VGA_R, VGA_G, VGA_B} <= {color_number_zero[23:16],
color_number_zero[15:8], color_number_zero[7:0]};
                          else if (life == 3'd1) {VGA_R, VGA_G, VGA_B} <= {color_number_one[23:16],
color_number_one[15:8], color_number_one[7:0]};
                          else if (life == 3'd2) {VGA_R, VGA_G, VGA_B} <= {color_number_two[23:16],
color_number_two[15:8], color_number_two[7:0]};
                          else if (life == 3'd3) {VGA_R, VGA_G, VGA_B} <= {color_number_three[23:16],
color_number_three[15:8], color_number_three[7:0]};
                    end
      //draw bomb word
                    else if (xcoor >= 10'd200 && (xcoor <= 10'd200 + 10'd121) && (vcount >= 0) && (vcount
<= 10'd35)) begin
                          bombword_address <= (xcoor-10'd200)+(vcount)*121;
                          {VGA_R, VGA_G, VGA_B} <= {color_bomb[23:16], color_bomb[15:8],
color_bomb[7:0]};
                    end
      //draw bomb number
                    else if (xcoor >= 10'd322 && (xcoor <= 10'd322 + 10'd31) && (vcount >= 0) && (vcount
<= 10'd31)) begin
                          number_address <= (xcoor-10'd322)+(vcount)*32;
                          if (bomb_value == 2'd0) {VGA_R, VGA_G, VGA_B} <= {color_number_zero[23:16],
color_number_zero[15:8], color_number_zero[7:0]};
                          else if (bomb_value == 2'd1) {VGA_R, VGA_G, VGA_B} <= {color_number_one[23:16],
color_number_one[15:8], color_number_one[7:0]};
                    end
      //draw score word
                    else if (xcoor >= 10'd400 && (xcoor <= 10'd400 + 10'd126) && (vcount >= 0) && (vcount
<= 10'd31)) begin
                          scoreword_address <= (xcoor-10'd400)+(vcount)*127;
                          {VGA_R, VGA_G, VGA_B} <= {color_score[23:16], color_score[15:8],
color_score[7:0]};
                    end
      //draw score number
```

```
                    else if (xcoor >= 10'd522 && (xcoor <= 10'd522 + 10'd31) && (vcount >= 0) && (vcount
<= 10'd31)) begin
                          number_address <= (xcoor-10'd522)+(vcount)*32;
                          if (tens == 4'd0) {VGA_R, VGA_G, VGA_B} <= {color_number_zero[23:16],
color_number_zero[15:8], color_number_zero[7:0]};
                          if (tens == 4'd1) {VGA_R, VGA_G, VGA_B} <= {color_number_one[23:16],
color_number_one[15:8], color_number_one[7:0]};
                          if (tens == 4'd2) {VGA_R, VGA_G, VGA_B} <= {color_number_two[23:16],
color_number_two[15:8], color_number_two[7:0]};
                          if (tens == 4'd3) {VGA_R, VGA_G, VGA_B} <= {color_number_three[23:16],
color_number_three[15:8], color_number_three[7:0]};
                          if (tens == 4'd4) {VGA_R, VGA_G, VGA_B} <= {color_number_four[23:16],
color_number_four[15:8], color_number_four[7:0]};
                          if (tens == 4'd5) {VGA_R, VGA_G, VGA_B} <= {color_number_five[23:16],
color_number_five[15:8], color_number_five[7:0]};
                          if (tens == 4'd6) {VGA_R, VGA_G, VGA_B} <= {color_number_six[23:16],
color_number_six[15:8], color_number_six[7:0]};
                          if (tens == 4'd7) {VGA_R, VGA_G, VGA_B} <= {color_number_seven[23:16],
color_number_seven[15:8], color_number_seven[7:0]};
                          if (tens == 4'd8) {VGA_R, VGA_G, VGA_B} <= {color_number_eight[23:16],
color_number_eight[15:8], color_number_eight[7:0]};
                          if (tens == 4'd9) {VGA_R, VGA_G, VGA_B} <= {color_number_nine[23:16],
color_number_nine[15:8], color_number_nine[7:0]};
                    end
      //draw score number
                    else if (xcoor >= 10'd554 && (xcoor <= 10'd554 + 10'd31) && (vcount >= 0) && (vcount
<= 10'd31)) begin
                          number_address <= (xcoor-10'd554)+(vcount)*32;
                          if (digits == 4'd0) {VGA_R, VGA_G, VGA_B} <= {color_number_zero[23:16],
color_number_zero[15:8], color_number_zero[7:0]};
                          if (digits == 4'd1) {VGA_R, VGA_G, VGA_B} <= {color_number_one[23:16],
color_number_one[15:8], color_number_one[7:0]};
                          if (digits == 4'd2) {VGA_R, VGA_G, VGA_B} <= {color_number_two[23:16],
color_number_two[15:8], color_number_two[7:0]};
                          if (digits == 4'd3) {VGA_R, VGA_G, VGA_B} <= {color_number_three[23:16],
color_number_three[15:8], color_number_three[7:0]};
                          if (digits == 4'd4) {VGA_R, VGA_G, VGA_B} <= {color_number_four[23:16],
color_number_four[15:8], color_number_four[7:0]};
                          if (digits == 4'd5) {VGA_R, VGA_G, VGA_B} <= {color_number_five[23:16],
color_number_five[15:8], color_number_five[7:0]};
                          if (digits == 4'd6) {VGA_R, VGA_G, VGA_B} <= {color_number_six[23:16],
color_number_six[15:8], color_number_six[7:0]};
                          if (digits == 4'd7) {VGA_R, VGA_G, VGA_B} <= {color_number_seven[23:16],
color_number_seven[15:8], color_number_seven[7:0]};
                          if (digits == 4'd8) {VGA_R, VGA_G, VGA_B} <= {color_number_eight[23:16],
color_number_eight[15:8], color_number_eight[7:0]};
                          if (digits == 4'd9) {VGA_R, VGA_G, VGA_B} <= {color_number_nine[23:16],
color_number_nine[15:8], color_number_nine[7:0]};
                    end
      //draw the title of the game
                    else if (xcoor >= 10'd0 && (xcoor <= 10'd31) && (vcount >= 80) && (vcount <=
10'd80+10'd259)) begin
                          startwarsword_address <= (xcoor)+(vcount-10'd80)*32;
                          {VGA_R, VGA_G, VGA_B} <= {color_starwars[23:16], color_starwars[15:8],
color_starwars[7:0]};
                    end
      //draw the start game
                    else if (xcoor >= 10'd32 && (xcoor <= 10'd32+10'd555) && (vcount >= 240) && (vcount <=
10'd240+10'd47)&&game_over) begin
                                 startword_address <= (xcoor-10'd32)+(vcount-10'd240)*556;
                                 {VGA_R, VGA_G, VGA_B} <= {color_start[23:16], color_start[15:8],
color_start[7:0]};
                    end
      //use line buffer to update all the entities.
                    else if ( xcoor >= 10'd32 && xcoor <= 10'd639 && vcount >=32 && !game_over) {VGA_R,
VGA_G, VGA_B} <= {mem_out[23:16], mem_out[15:8], mem_out[7:0]};
                    else {VGA_R, VGA_G, VGA_B} <= {8'd0,8'd0,8'd0};
                          draw_states <= draw_waiting;
              end
       end

  //set the output of drawing line buffer to
  //mem_out, set the output of updating line
  //buffer to mem_out_u
  always_comb begin
    mem_out = 32'd0;
    if (line_buffer_select == uA_cB_dC) begin
             mem_out = C_out;
             mem_out_u = A_out;
       end
    else if (line_buffer_select == uB_cC_dA) begin
             mem_out = A_out;
             mem_out_u = B_out;
       end
    else begin
             mem_out = B_out;
             mem_out_u = C_out;
       end
      end

      //rom used to draw sprites.
  rom_starwarsword starwars (.address(startwarsword_address),.clock(clk50),.q(color_starwars));
  rom_startgameword startword (.address(startword_address),.clock(clk50),.q(color_start));
      rom_scoreword scoreword (.address(scoreword_address),.clock(clk50),.q(color_score));
      rom_bombword bombword(.address(bombword_address),.clock(clk50),.q(color_bomb));
      rom_0 number_zero(.address(number_address),.clock(clk50),.q(color_number_zero));
      rom_1 number_one(.address(number_address),.clock(clk50),.q(color_number_one));
      rom_2 number_two(.address(number_address),.clock(clk50),.q(color_number_two));
      rom_3 number_three(.address(number_address),.clock(clk50),.q(color_number_three));
      rom_4 number_four(.address(number_address),.clock(clk50),.q(color_number_four));
      rom_5 number_five(.address(number_address),.clock(clk50),.q(color_number_five));
      rom_6 number_six(.address(number_address),.clock(clk50),.q(color_number_six));
      rom_7 number_seven(.address(number_address),.clock(clk50),.q(color_number_seven));
      rom_8 number_eight(.address(number_address),.clock(clk50),.q(color_number_eight));
      rom_9 number_nine(.address(number_address),.clock(clk50),.q(color_number_nine));
   rom_lifeword lifeword(.address(lifeword_address),.clock(clk50),.q(color_lifeword));
      rom_spaceship spaceship(.address(space_address),.clock(clk50),.q(color_spaceship));
```

```
        rom_background background(.address(background_address),.clock(clk50),.q(color_background));
        rom_bullet1 bullet(.address(bullet_rom_address),.clock(clk50),.q(color_bullet));
        rom_spawn0 spawn0(.address(spawn_address),.clock(clk50),.q(color_spawn0));
        rom_spawn45 spawn45(.address(spawn_address),.clock(clk50),.q(color_spawn45));
        rom_enemy4 enemy400(.address(enemy4_address),.clock(clk50),.q(color_enemy400));
        rom_enemy200 enemy500(.address(enemy5_address),.clock(clk50),.q(color_enemy500));
        rom_enemy6 enemy600(.address(enemy6_address),.clock(clk50),.q(color_enemy600));
        rom_enemy700 enemy700(.address(enemy7_address),.clock(clk50),.q(color_enemy700));
        rom_explosion_bullet
explosion_bullet(.address(explosion_bullet_address),.clock(clk50),.q(color_explosion_bullet));
        rom_explosion_bullet180
explosion_bullet180(.address(explosion_bullet_address),.clock(clk50),.q(color_explosion_bullet180));
endmodule // VGA_LED_Emulator
```

6.VGA_BALL.sv

```
/* This module is the connection between the software and vga_emulator
*  It takes the messages sends from software, reorganize and store the message[0]to message[255]
*  into[31:0]data1[0:63] or [31:0]data2[0:63] alternately
*  The VGA_Emulator module takes the data from either data1 or data2 as the information to draw
*
*  Zhenyu Zhu, Fang Fang, Columbia University
*/
module VGA_BALL( input logic clk,
    input logic reset,
    input logic [15:0] writedata,
    input logic [7:0] address,
    input logic write,
    input logic chipselect,
    output logic VGA_audio_bullet,VGA_audio_collision,
    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);

        logic draw_done; //emulator finishes its work
        logic update_done;//emulator could start work
        logic enable_ram;//ram in emluator could start receive info
        logic ram_select;//choose which ram is used to store info
        logic end_writedata_to_data;
        logic end_data_to_emulator;
        logic [1:0] e_counter2;   //added logic for sending data to emulator in multiple cycles
        logic [7:0] e_counter3;  //reset state counter
        logic en_emulator;      //enable the ram in Emulator to store the data
        logic [1:0] life;
        logic [1:0] bomb_value;
        logic [9:0] score;
        logic [3:0] tens;   // the decade digit for the score
        logic [3:0] digits;  // the unit digit for the score
        logic game_over;   // indicating the lifes of the avatar have been used up and restart the game
        reg [9:0] data1 [0:255];//reg used to store info
        reg [9:0] data2 [0:255];//reg used to store info
        logic [31:0] data_to_emulator;
        logic [31:0] data_to_emulator_tmp;
        logic [9:0] data_index;
```

```
	logic [31:0] A;
	logic [31:0] B;
	logic [31:0] C;
	logic [31:0]A_in;
	logic [31:0]B_in;
	logic [31:0]C_in;
	logic [31:0] mem_out_u,mem_out,ram_out;
	//states makes this work
	enum logic [2:0] {reset_states, processing} states;
	enum logic[3:0] {idle, wait_stable, process_data, transfer_data} sub_states;

	VGA_BALL_Emulator ball_emulator(.*,.clk50(clk), .draw_done(draw_done),
.update_done(update_done),.enable_ram(enable_ram),.ram_select(ram_select),.game_logic_to_emulator(data_to_
emulator), .A(A), .B(B), .C(C));

	always_ff @(posedge clk) begin
	    if (reset) begin
			ram_select <= 0;
			data_to_emulator <= 0;
			data_index <= 0;
			enable_ram <= 1;
			update_done <= 1;
			e_counter1 <= 0;
			e_counter2 <= 0;
			e_counter3 <=0;
			states <= reset_states;
			game_over <= 1;
		end
		//give the intial value to emulator
		else if (states == reset_states)begin
			if (e_counter3== 8'd0) begin
				data1[e_counter3] <= 10'd1;
				data2[e_counter3] <= 10'd1;
				e_counter3 <= e_counter3 + 1;
			end
		  else if (e_counter3 == 8'd1)begin
				data1[e_counter3] <= 10'd320;
				data2[e_counter3] <= 10'd320;
				e_counter3 <= e_counter3 + 1;
		  end
		  else if (e_counter3 == 8'd2)begin
				data1[e_counter3] <= 10'd240;
				data2[e_counter3] <= 10'd240;
				e_counter3 <= e_counter3 + 1;
		  end
		  else if (e_counter3 == 8'd3)begin
			data1[e_counter3] <= 10'd0;
				data2[e_counter3] <= 10'd0;
				e_counter3 <= 0;
			states<=processing;
			sub_states <= wait_stable;
		  end
		end
		//Start to receive info from software and
```

```
            //tranfer the data to VGA_BALL_Emulator
            else if (states == processing) begin
                  if (chipselect && write) begin
                        if (address <= 8'd255) begin
                              if(address == 8'd240) life <= writedata;
                              else if (address == 8'd255) game_over <= writedata;
                              else if (address == 8'd248) digits <= writedata;
                              else if (address == 8'd247) tens <= writedata;
                              else if(address == 8'd241) score <= writedata;
                              else if(address == 8'd242) bomb_value <= writedata;
                              //when ram_select is 0, store the writedata into data1
                              if(ram_select == 0) data1[address] <= writedata;
                              //when ram_select is 1, store the writedata into data2
                              else data2[address] <= writedata;
                              if(address ==  8'd255) end_writedata_to_data <= 1;
                        end
                  end
                  ////end of software data transmission
                  else end_writedata_to_data <= 1;

                  //added counter to control multiple cycles to send data to emulator
                  if(enable_ram==1) begin
                        if(data_index!=10'd64)begin
                              //fetch data from data1 or data 2 and store it in tem reg
                              if(sub_states==wait_stable)begin
                                    if(e_counter2==0)begin
                                          data_to_emulator_tmp[31:24] <=
ram_select?data1[4*data_index]:data2[4*data_index];
                                          e_counter2 <= e_counter2 + 1;
                                    end
                                    else if(e_counter2==1)begin
                                          data_to_emulator_tmp[23:14] <=
ram_select?data1[4*data_index+1]:data2[4*data_index +1];
                                          e_counter2 <= e_counter2 + 1;
                                    end
                                    else if(e_counter2==2)begin
                                          data_to_emulator_tmp[13:4] <=
ram_select?data1[4*data_index+2]:data2[4*data_index+2];
                                          e_counter2 <= e_counter2 +1;
                                    end
                                    else if(e_counter2==3)begin
                                          data_to_emulator_tmp[3:0] <=
ram_select?data1[4*data_index+3]:data2[4*data_index+3];
                                          e_counter2 <= 0;
                                          sub_states <= transfer_data;
                                    end
                              end
                              //start to transfer the data to VGA_BALL_Emulator
                              else if (sub_states == transfer_data)begin
                                    data_to_emulator<=data_to_emulator_tmp;
                                    en_emulator <= 1;   //enable the ram in the VGA_BALL_Emulator to
store the data
                                    sub_states <= process_data;
                                    //receive audio info set flags or clear flags
```

```
                                        if (data_index == 10'd61)begin
                                              VGA_audio_bullet <= data_to_emulator_tmp[14];
                                              VGA_audio_collision <= data_to_emulator_tmp[24];
                                        end
                                  end
                                  //keeping fetching data and sending it out until it finsihes transmitting
64 objects
                                  else if(sub_states==process_data)begin
                                        VGA_audio_bullet <= 0;
                                        VGA_audio_collision <= 0;
                                        en_emulator <= 0;
                                        data_index <= data_index + 1;
                                        sub_states <= wait_stable;
                                  end
                            end
                            else begin
                                  data_index <=0;
                                  end_data_to_emulator <= 1;
                                  enable_ram <= 0;
                            end
                      end
                      //when finishing receiving 256 messages from the software as well as finishing
transmitting
               // 64 objects to the emulator, wait the emulator to finish its drawing
                      //and restart emulator to draw.
                      if(end_data_to_emulator & end_writedata_to_data)begin
                            if(draw_done)begin
                                  update_done <= 1;  // indicating the emulator could restart to draw from
(0,0)
                       /*toggle the select bit to alternate the updating reg(from the software) and the
transmitting reg (to the Emulator)*/
                                  ram_select <= ~ram_select;
                                  end_data_to_emulator <= 0;
                                  end_writedata_to_data <=0;
                                  enable_ram <= 1;
                            end
                      end
                      else update_done <= 0;

                end
          end

endmodule
```