Remote Pong Game
Final Report

CSEE 4840 Embedded System Design

Fengyi Song Juchao Zhang Mingrui Liu Wanding Li
{fs2523, jz2606, ml3663,wl2501}@columbia.edu

May 14, 2015

Contents

1. Overview

In this project, we design and implement a remote Pong game on the FPGA SoCKit development board. Simple Break Pong game is a popular video game where the player uses a paddle to bounced a ball and then the ball might run into a brick and the brick will disappear. The theme of our game will be based on the top universities in the U.S. Each brick represents a specific university and the paddle represents Columbia University. Once the other universities are bounced by a ball, it will disappear but Columbia will not. The game will generate several bricks on the screen and the player will bounce bricks using an on screen paddle with a wiimote controller. Each player of the game has three lives. Once the ball runs below the paddle, one life loses. The entire game is won when the player completes bounced all the bricks except Columbia brick successfully. Figure 1 is a snapshop of the start screen of the Simple Brick Breaker video game. The score and lives can be seen at the very top of the screen. The ball at the center will bounce the bricks in the first half of the screen once the game starts

.

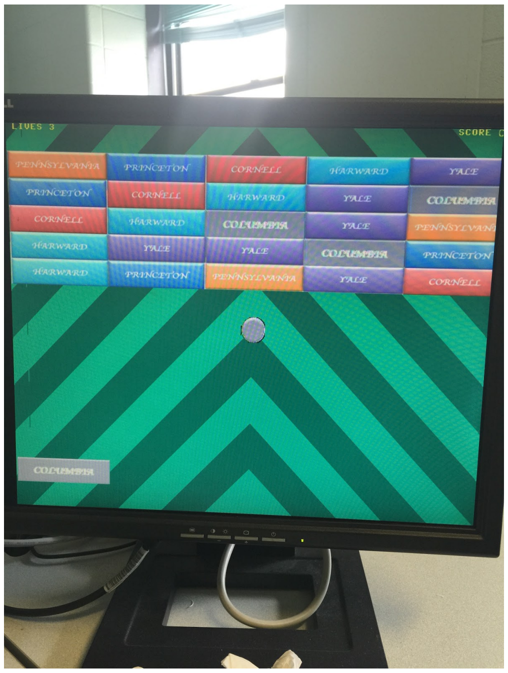

Figure 1: game start screen

2. High Level Design

The primary components that make up our game includes the game logic, device drivers, wiimote controller for input control, audio controller, the display module that includes the sprite and VGA controller, and a data storage module that includes on-chip ROM for the audio and image files as well as HPS SDRAM for our software code (see Figure 2). The game logic module interfaces with several of the other modules in the game including the wiimote 3 controller as well as the device drivers in order to control the on and off of each brick. The game logic controls the progression of the entire game from start to end based on the defined game rules. The game includes several sprites for background, score, life, ball and bricks. The sprites in addition to the audio files utilize a large amount of ROM space on the FPGA and so are carefully designed to efficiently use the available logic on the FPGA. Each of the components in our game design will be discussed in detail below.

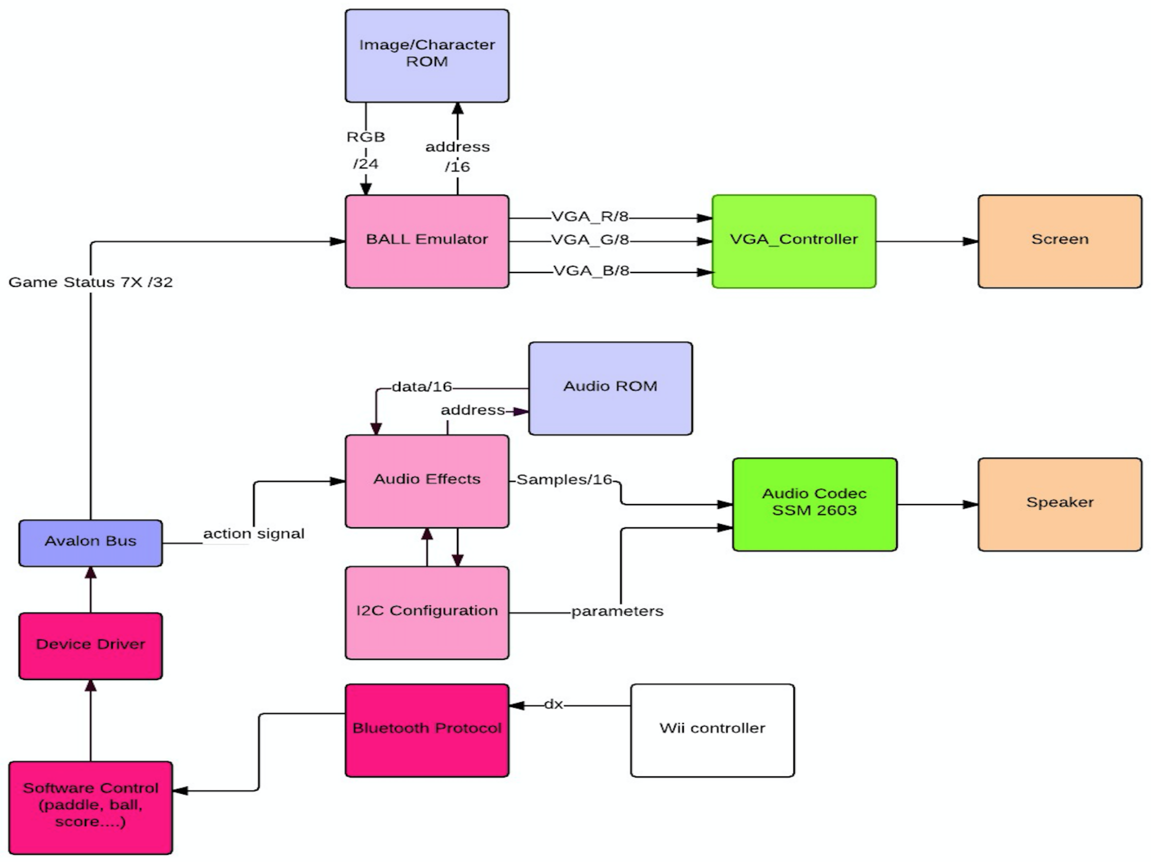

Figure 2: Overview of Game

3. Graphs and Audio Preparation

The preparations required for the graphics and audio are similar. Since we need to store the audio file and picture files in ROM blocks, firstly we need to search online music files and picture files. Then we need to use Matlab to generate MIF format files for them.

3.1. Graphs Preparation

There are only two different kinds of pictures we used in our game. The first kind of picture is different bricks representing different top universities(See Figure 3). The size of each brick is 128*32. Since the size of the screen is 640*480, we can layout 5 bricks each row and we only layout 5 bricks each column to make room for the ball to run. Then we need a ball to bounce bricks. Since the ROM block in quartus can only store a square-shape picture, we choose a square of size 32*32 with the corner of the square is black. So when we display the ball, we add a condition to it. If the VGA of the ball is 0, which means it is scanning the corner of the ball, we do not display that. Then we use MATLAB to generate the .mif format files and use them as the initializable files for ROM blocks. Except bricks and balls, we also have our background, lives and score. We generate our own .mif files only using ‘’1’’ and ‘’0’’ and then draw them out by ourselves.

Figure 3: Brick

3.2. Audio Configuration

The audio preparation is very similar to that of graphs. First we need to find two music. One is used when the ball is bounced onto a brick and the other is used when the ball is bounced onto the paddle. Then we need to use MATLAB to sample the sound and get the .mif files of the sound. Then we use .mif files as the initializable files for sound ROMs.

4. Wii Controller

We are using Wii controller to move the paddle. To connect the wii controller with our device, we are using a Bluetooth USB dongle connecting to the board. Also since we need to move wii handler, we need a sensor bar to as the base of the distance. The pictures of wiimote controller, sensor bar and Bluetooth USB dongle are shown in Figure 4. There is a camera in front of the wiimote handler. The camera can sense the relative position with the sensor bar. Then the wiimote controller will send the position information to the board through bluetooth dongle.

Since there is no wiimote driver and bluetooth driver, first we need to install wiimote driver from wii open source and blueZ. We use BlueZ [1] as the Bluetooth stack to communicate between the Wiimote and Linux host. BlueZ[1] is an official Linux Bluetooth protocol stack. It provides support for the core Blutooth layers and protocols. It can program an implementation of the Bluetooth wireless stanards specifications for Linux. libwiimote [2] is a C-library build on BlueZ that provides a simple API for communicating between the Wiimote and the Linux host. It can read accelerometer data from the Wiimote. We can get the data of the camera of Wiimote by calling functions provided by libwiimote directly and save huge effort of doing nasty math computations. In this project, we use BlueZ and libwiimote together to make the developing of Wii Controller module easier. The first sensor bar we use is a regular sensor bar used for wii game. However, we find that it is hard to operate because there are 6 spots in the sensor bar while we only need one spot for getting controller's position. So, we change with another sensor bar which looks like a laser pen. While searching for controller's signal, we have to hold the red button in the controller.

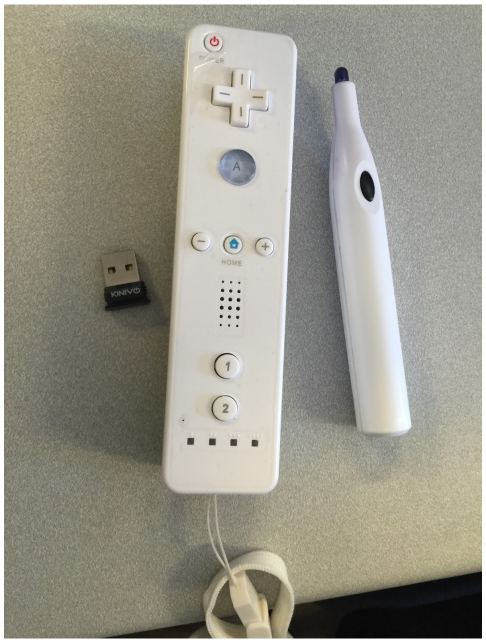
Figure 4: USB dongle, wiimote handler and sensor bar

5. Game Logic

We implement game logic in software using C programming which is a user space program. There are mainly three different types of functions game logic should implement. The first function is that it should provide an API to VGA driver. Since we need to write the coordinate of the ball, paddle to hardware, we need to provide functions to write them to device driver first and driver then transfer them to hardware. Besides, since once the ball bounced the brick, the brick will vanish. We need to add a signal caller on which is a 32-bit integer. Each bit of the on signal represents the on and off of a brick. If that bit is 1, then the brick will not vanish. The most important function is the movement of ball and paddle. We have a while(1) loop in main function. In the while (1) loop, we increment the coordinate of the ball very 7000ns by adding uspleep(7000) in the end. And every time the coordinate of the ball increments, we should check whether the brick is inside the ball or not. If the brick is inside the ball, we will send a signal to hardware and then make that brick vanish. At the same time we need to write paddle coordinate into hardware. So we also have a pthread which is independent of while(1) loop in main function and they share the coordinate of paddle. The structure of the game logic is shown in figure 5.

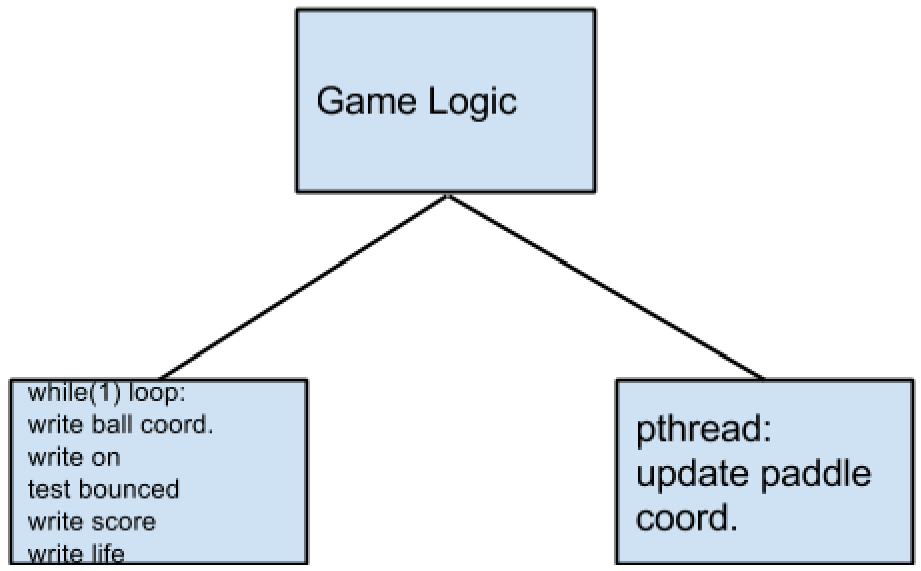

Figure 5: Game logic block

6. Device Drivers

Device driver is mainly for data transfer. We use iowrite functions in driver to write score, life, on, coordinates to hardware. The VGA device driver is similar to the one used in Lab 3. Our device driver uses several ioctl calls to write to the memory-mapped VGA device. This memory is accessed by the FPGA using the avalon bus. The FPGA uses 3-bit address bits to access 7 locations that store 32-bit data. The data written to peripheral memory-mapped device using the device driver include the x, y positions of the ball sprites. Similar to Lab 3, the driver code uses extensive bound checking to avoid any out-of-bounds errors. The Ioctl calls are only used to write to the memory-mapped device and does not involve any read ioctl calls.

7. Sprite and VGA Display

To save memory, for the background, game information and gameover interface, we represent each pixel by binary, and assign RGB to every pixel after reading the binary information from memory. As to the background interface, there are only 2 colors, so binary is sufficient to distinguish pixels in terms of colors. For gameover interface, we assign 1 to pixels displaying letter and 0 to void portion in sprites, and assign colors to different zones after reading the binary information from the rom.

For the bricks, the paddle and the ball, since they are all small, we store color info into memory by transferring them to .mif file using Matlab.

For the background part, since everything shown in the background is linear, we tried to use slopes between pixels to distinguish different zones. However, there were quite many noisy points accompany with the background, since the CPU on Sockit cannot compute the corresponding zone of every pixel in such high frequency as 25MHz.

Then we decided to decompose the background into 6 sprites(two 20 * 120 sprites and four 60 * 120 sprites). And read from rom one bit right next to the need-to-be-displayed bit to handle delay. It works out well.

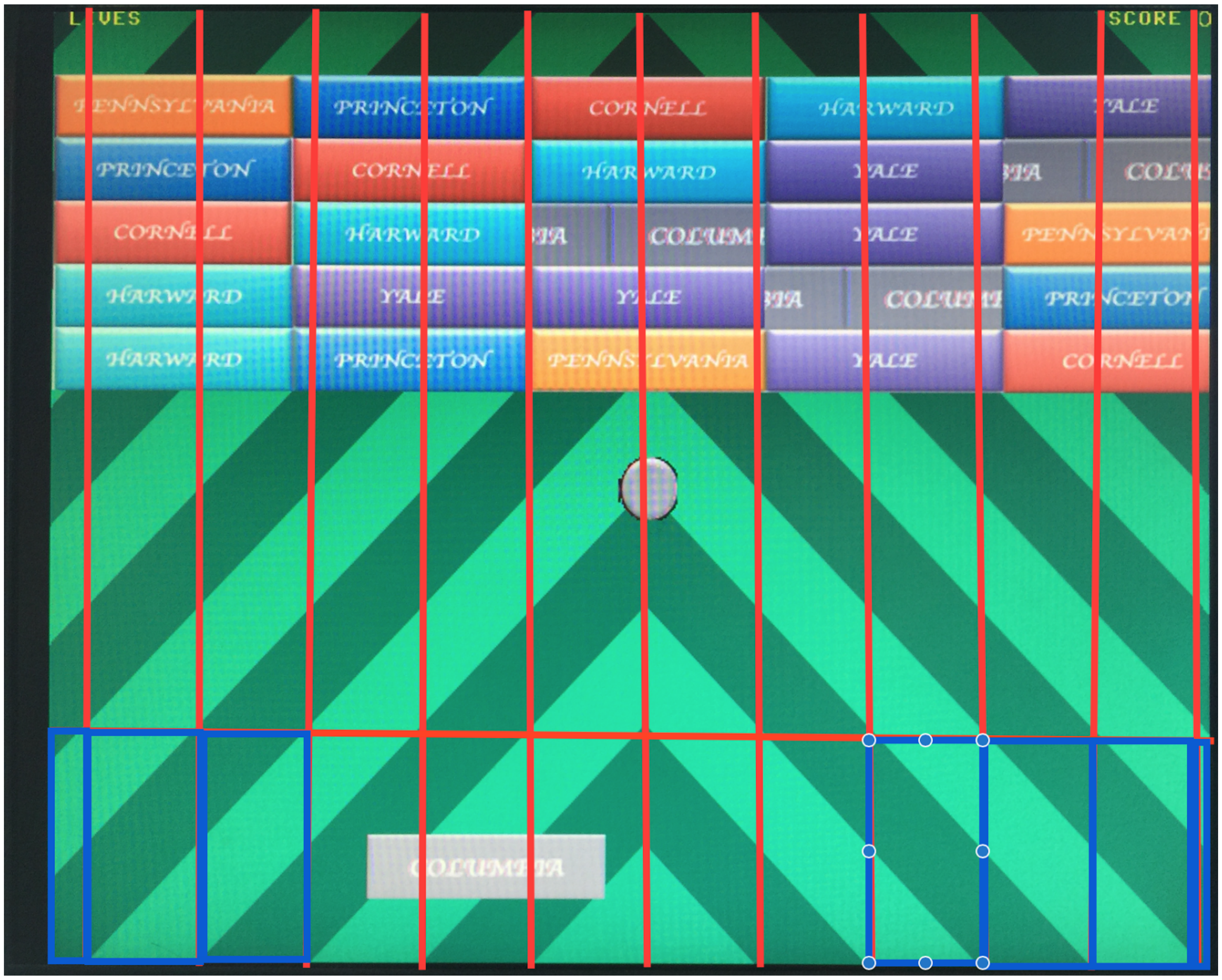

Figure 6: Background partition

We can see from the above figure that every column consists of a single sprite. We align every sprite horizontally and repeat the sprite vertically.
For the gameover interface, we define different zones for each individual letter, so as to assign different colors to different letters. The interface is shown as below:

Figure 7: Ending picture

For the game interface, we use software to tell hardware what to display and where to display. For example, software will compute the xlocation and ylocation and store it in a specific memory, then hardware will read from this memory, get the location info and display the ball. Also, software will tell the current score and lives hardware should display, and not to display the brick when it is hit by the ball.

The size of each sprite is shown in the table below:

name	brick	ball	Background_s	Background_l	gameover	info
size	128*32*24	32*32*24	20*120	60*120	64*64	8*8
number	6	1	2	4	7	36

Memory info for every sprite

8. Audio Controller

We are using audio codec chip on the Sockit board. In order to drive the codec chip, it needs a I2C protocol and Audio Codec module to configure SSM 2603. The audio effects are triggered based on the action of the paddle or the moment when the bricks got hit. Therefore, we need two signal asserted from the software control passing into the FPGA board through Avalon bus.

Before synchronization with software, a proper configuration and protocol need to be developed. Referring to Howard Mao's tutorial, we develop three modules to drive the Audio Codec SSM 2603. The first one is Audio Codec which configures the parameters that the chip needs to output the sounds signals such as Volume(0 dB), mode (Slave), Sampling Frequency(11.3 Mhz), Power on/off. The second is I2C protocol. Based on the reference from Howard Mao's code, we develop our own audio interface utilizing the I2C protocol. In the module of audio_effects, we access two samples that we stored before. Each time when sample request asserted, the module accessed the rom block. When reaching the end of the audio ROM, the data became 0. In addition, there are two clocks derived from audio clock. One is channel clock which sends sample on one channel at a time. The second is bit clock which sends one bit of each sample each request.

9. Experience and Issues

We encountered both hardware issues and software issues. In the hardware part, the biggest issue is our algorithms of background plotting is too complex. We couldn't finish one calculation during a clock period. The result of that is the background may have a lot of noises, which makes it less professional for a game. We solved this problem by simplify our algorithms. The techniques we used are to divide the whole screen into a few parts. Since our background is made up of iterative sprites, we create a single .mif file which contains all the sprites information. So when we plot the background, we call different parts of the mif files iteratively. Since there is only two colors in background, 0 and 1 can represent these two color perfectly. Besides, since we are using sequential logic in our design, there is one clock delay shown in the screen. To remove the delay, when we read address of the ROM, we always read the address+1 so that when it is supposed to read address+1, the screen shows the address pixel.

The software issues we faced are very similar to each other. First of all, we are using bluetooth to connect wiimote controller to FPGA board. However, there is no bluetooth driver inside FPGA Linux kernel. We have to update the version of the Linux kernel and file system. Then we need to connect FPGA to the internet and use apt-get to install bluetooth driver and header files. The other issue is that we do not have wiimote controller driver which is available online. We did the same things as we installed bluetooth driver. The other issue is that when we can connect wiimote controller correctly, the movement of ball is not smooth and it seems whenever the ball is about to be bounced by the paddle, it will slow down its pace. To fix this problem, we analyzed our code and found that we update the coordinates of ball and paddle in the same while(1) loop, which means they are updating information in the same thread. In this thread, when updating the coordinates of paddle, we need to write into that variable. However, when we want to check the position of ball and paddle, we must read that variable. Once reading and writing happens together, there would be problems. So we write one more thread where we only set coordinate of paddle and add a mutex lock to it. Thus, reading and writing have to happen simultaneously and the conflict would be canceled.

10. Lesson Learned

There are a few lessons we learned from this project. In the aspect of hardware, we learned how to avoid clock delay in the synchronous system. Besides, we learned that we have to make the algorithms simple in case that one calculation will not be finished in one clock cycle.

In the aspect of software, we learned how to integrate device driver and user space programs as well as how to transfer data between device driver and hardware. And we also learned how to avoid reading and writing at the same time.

In total, we have a deeper understanding of how hardware and software systems works and how to interact with software and hardware.

11. Future Work

We can further our work in a few aspects. The first aspect is game logic. Right now our game logic is very simple. Each player only has three lives, if the player pass the life, the ball will accelerate. If the player fail the life, the ball keeps its speed. The game ends once all these three lives loses. We can make more interesting. For example, we can add multi-player mode and show the rank of each player.

The second aspect is wiimote controller. Since wiimote handler has a camera in front of it which can capture 6 spots and we only use 1-spot dongle now, in the future we can use a 6-spot dongle to make the movement of paddle more accurate. Besides, we are using a very simple scaling algorithm to scale the relative position to VGA screen range. We cannot move wiimote handler in a large angle. We can improve the scaling algorithms to increase the user experience.

The third aspect is to add more interesting features to bricks. For example, we can add surprise bricks once bounced, it will drop candy.

In total, there is still a lot to improve our game. However, for our project, we used both hardware interface and software interface and practiced a lot of embedded system skills. All the improvements will be based on our current game.

12. Contributions

Fengyi Song	Audio effect
Junchao Zhang	Wii controller, bluetooth
Mingrui Liu	Game logic, Linux driver
Wanding Li	Sprite controller

13. Milestones

Milestone	Date	Goal	Accomplishment
Milestone1	2015/4/2	basic game display, score increase, brick vanish	basic game display
Milestone2	2015/4/14	paddle can be controller by wiimote controller	paddle can connect via bluetooth but not moving smoothly
Milestone3	2015/4/28	adding sound effects	added sound effects
Deadline	2015/5/14	finish the project, report and presentation	All achieved!

14. References

[1] BlueZ, “Official linux bluetooth protocol stack.” http://www.bluez.org.
[2] libwiimote, “Simple wiimote library for linux.” http://libwiimote.sourceforge.net.

15. C code

lab3-sw/hello.c

```
/*
 * Userspace program that communicates with the led_vga device driver
 * primarily through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 */

#include <stdio.h>
#include "vga_led.h"
#include "wiicontroller.h"
#include "wiimote.h"
#include "wiimote_api.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <sndfile.h>
#include <stdbool.h>
#include <pthread.h>

pthread_t paddle_thread;
void *paddle_thread_f(void*);
pthread_mutex_t lock;
int vga_ball_fd;
unsigned int ppx;

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

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

/* Write the contents of the array to the display */
void write_segments(const unsigned char segs[8])
{
  vga_ball_arg_t vla;
  int i;
  for (i = 0 ; i < VGA_LED_DIGITS ; i++) {
    vla.digit = i;
    vla.segments = segs[i];
    if (ioctl(vga_ball_fd, VGA_LED_WRITE_DIGIT, &vla)) {
      perror("ioctl(VGA_LED_WRITE_DIGIT) failed");
      return;
    }
  }
}
void write_coords(unsigned int x, unsigned int y) {
circle_center cc;
cc.x = x;
cc.y = y;
if(ioctl(vga_ball_fd, VGA_LED_WRITE_CENTER, &cc)) {
perror("ioctl(VGA_LED_WRITE_CENTER) failed");
return;
}
}

void write_pad(unsigned int x) {

if (x < 0 || x > 700) return;
if(x>=512)
     x = 512;
ppx=512-x;
```

```
//printf("write_pad...\n");
if(ioctl(vga_ball_fd, VGA_LED_WRITE_PAD, &ppx)) {
perror("ioctl(VGA_LED_WRITE_PAD) failed");

return;
}
}

void write_score(unsigned int x) {
int score = x;
//printf("write_pad...\n");
if(ioctl(vga_ball_fd, VGA_LED_WRITE_SCORE, &score)) {
perror("ioctl(VGA_LED_WRITE_SCORE) failed");
return;
}
}

void write_life(unsigned int x) {
int life = x;
//printf("write_pad...\n");
if(ioctl(vga_ball_fd, VGA_LED_WRITE_LIFE, &life)) {
perror("ioctl(VGA_LED_WRITE_LIFE) failed");
return;
}
}

void write_on(unsigned int x) {
unsigned int on = x;
if(ioctl(vga_ball_fd, VGA_LED_WRITE_ON, &on)) {
perror("ioctl(VGA_LED_WRITE_ON) failed");
return;
}
}

void write_audio(short x) {
int audio = x;
if(ioctl(vga_ball_fd, VGA_LED_WRITE_AUDIO, &audio)) {
perror("ioctl(VGA_LED_WRITE_AUDIO) failed");
return;
}
}

int bounce_columbia(int row, int col, int x, int y, int dx, int dy){
        if((y==(row+1)*32-16 || y==(row+2)*32+16)&& x>=col*128 &&
x<=(1+col)*128)
                return 1;
        else if((x==col*128-16 || x==(col+1)*128+16) && y>=(row+1)*32
&& y<= (row+2)*32)
                return 2;
        else
if((((row+1)*32-y)*((row+1)*32-y))+(col*128-x)*(col*128-x)>=16*16 &&
dx==1 && dy==1)
                return 3;
        else
if((((row+1)*32-y)*((row+1)*32-y))+((col+1)*128-x)*((col+1)*128-x)>=1
6*16 && dx==-1 && dy==1)
                return 3;
        else
if((((row+2)*32-y)*((row+2)*32-y))+(col*128-x)*(col*128-x)>=16*16 &&
dx==1 && dy==-1)
                return 3;
        else
if((((row+2)*32-y)*((row+2)*32-y))+((col+1)*128-x)*((col+1)*128-x)>=1
6*16 && dx==-1 && dy==-1)
                return 3;
        else return 0;

}
int main()
{
  vga_ball_arg_t vla;

  if(pthread_mutex_init(&lock,NULL)!=0)
 {
      printf("mutex init failed\n");
      return 1;
  }
  int i;
  static const char filename[] = "/dev/vga_ball";

  static unsigned char message[8] = { 0x39, 0x6D, 0x79, 0x79,
                            0x66, 0x7F, 0x66, 0x3F };

  printf("VGA LED 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: ");
  int x = 320;
  int y = 240;
  int dx = -1;
  int dy = -1;
  unsigned int on = 0xffffffff;
  write_on(on);
  printf("on is %d\n", on);


  int score = 0;
  printf("score is %d\n", score);
  write_score(score);
  int life = 3;
  write_life(life);



//read wave file --------------------------
SNDFILE *sf;
SF_INFO info;
int num_channels;
int num, num_items;
short *buf;
int f,sr, c;
int k, j;
FILE *out;
info.format = 0;
sf = sf_open("QQ.wav", SFM_READ, &info);
if(sf == NULL) {
     printf("Failed to open file\n");
     exit(-1);
     }
f= info.frames;
sr = info.samplerate;
c= info.channels;
printf("frames is %d\n",f);
printf("samplerate is %d\n",sr);
printf("channels is %d\n",c);
```

```
num_items=f*c;
printf("num_items is is %d\n",num_items);

//allocate space for data
buf = (short* ) malloc(num_items*sizeof(short));
num = sf_read_short(sf,buf,num_items);
sf_close(sf);
printf("read %d items\n",num);

//write data to filedata.out
out = fopen("filedata.out","w");



pthread_create(&paddle_thread, NULL, paddle_thread_f, NULL);

sleep(10);
//fclose(out);
k=0;
 while(1) {


     write_audio(buf[k]);
     printf("buf is %d\n",buf[k]);
     fprintf(out, "%d ", buf[k]);
     fprintf(out, "\n");
     if(k<num)
          k=k+c;


x += dx;
y += dy;




write_coords(x, y);
write_on(on);


if(x == 15|| x==640-16)
dx = -dx;

bool bounced=(y == 15) || (y == 480-32-32-16 && x>=(ppx) &&
x<=((ppx)+128));

//printf("px is %d x y %d %d\n", ppx,x,y);
if(bounced) {
//printf("px is %d x y %d %d\n", ppx,x,y);
dy = -dy;
}

//on = ~on;
int row, col,num;
num = 0;
for(row=0;row<=4;row++) {
      for(col=0;col<=4;col++) {

       if((on>>num)&1 == 1 && (y>=(row+1)*32-16 &&
y<=(row+2)*32+16)&& x>=col*128 && x<=(1+col)*128) {
                if(bounce_columbia(row,col,x,y,dx,dy)==1 && (num==9
|| num==18 ||num==12)) {
                      dy = -dy;
                      printf("columibia bounced y dx %d
dy%d\n",dx,dy);
                }
                else if(bounce_columbia(row,col,x,y,dx,dy)==3
&&(num==9 || num ==12 || num == 18)) {
                      dy=-dy;
                      printf("columibia bounced x y dx %d
dy%d\n",dx,dy);
                }
                else if(num!=9&&num!=12&&num!=18) {
                unsigned int mask = ~(1<<num);

                on = on&mask;

//              write_on(on);
                printf("on&(1<<num)==1<<num? %d\n",(on>>num)&1);
                printf("row is %d col is %d on&num is
%d\n",row,col,on&(1<<num));
                score++;
                printf("score is %d\n",score);
                write_score(score);
                dy = -dy;
//              write_coords(x, y);
                }
                }
       if((on>>num)&1==1 && (x>=col*128-16 && x<=(col+1)*128+16) &&
y>=(row+1)*32 && y<= (row+2)*32) {
                if(bounce_columbia(row,col,x,y,dx,dy)==2 &&(num==9 ||
num ==12 || num == 18)) {
                     dx= -dx;
                     printf("columibia bounced x dx %d
dy%d\n",dx,dy);
                }
                else if(bounce_columbia(row,col,x,y,dx,dy)==3
&&(num==9 || num ==12 || num == 18)) {
                     dx= -dx;
                     printf("columibia bounced x y dx %d
dy%d\n",dx,dy);
                }
                else if(num!=9&&num!=12&&num!=18) {
                unsigned int mask = ~(1<<num);

                on = on&mask;

//              write_on(on);
                printf("row is %d col is %d on&num is%d\n", row, col,
on&(1<<num));
                score++;
                printf("score is %d\n", score);
                write_score(score);
                dx = -dx;
//              write_coords(x, y);
                }
                }
                num++;
      }
}

if(y>480-16 && (x<(ppx)||x>(ppx)+128)) {
      life--;
      write_life(life);
      score = 0;
      write_score(score);
      on = 0xffffffff;
      write_on(on);
      x= 320;
      y= 240;
```

```
        write_coords(x,y);
        if(life==0) {
             printf("game over\n");
             return;
        }
        sleep(10);


}

 if(life == 2 && on==0xfe028200)
 usleep(5000);
else if(life == 1&& on==0xfe028200)
 usleep(3000);
else
 usleep(7000);
}
//wii_disconnect(wii);
//fclose(out);
  pthread_cancel(paddle_thread);

  /* Wait for the network thread to finish */
  pthread_join(paddle_thread, NULL);

  printf("current state: ");

  printf("VGA LED Userspace program terminating\n");
  return 0;
}

void *paddle_thread_f(void *ignored)
{

  unsigned int py;
  unsigned int px;


  wiimote_t wii;

  wii = wii_connect();
  while(1) {
pthread_mutex_lock(&lock);
  wii_getpos(&wii, &px, &py);
  printf("wii px is %d\n",px);

//  if(px>=0 && px<=640-128)
//    write_pad(512 - px);
  write_pad(px);

  usleep(7000);
  pthread_mutex_unlock(&lock);
  }

  return NULL;
}
```

lab3-sw/vga_led.c

```
/*
 * Device driver for the VGA LED Emulator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
 *
 * "make" to build
 * insmod vga_ball.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree 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_led.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 */
     u8 segments[VGA_LED_DIGITS];
} dev;

/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set
up
 */
static void write_digit(int digit, u8 segments)
{
     iowrite8(segments, dev.virtbase + digit);
     dev.segments[digit] = segments;
}

/*
 * 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;
     circle_center cc;
```

```
        int px;
        int on;
        int score;
        int life;
        int audio;
        switch (cmd) {
        case VGA_LED_WRITE_DIGIT:
             if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                           sizeof(vga_ball_arg_t)))
                  return -EACCES;
             if (vla.digit > 8)
                  return -EINVAL;
             write_digit(vla.digit, vla.segments);
             break;

        case VGA_LED_READ_DIGIT:
             if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                           sizeof(vga_ball_arg_t)))
                  return -EACCES;
             if (vla.digit > 8)
                  return -EINVAL;
             vla.segments = dev.segments[vla.digit];
             if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                         sizeof(vga_ball_arg_t)))
                  return -EACCES;
             break;
        case VGA_LED_WRITE_CENTER:
             if (copy_from_user(&cc, (circle_center *) arg,
    sizeof(circle_center)))
                  return -EACCES;
             iowrite32((cc.x << 10 | cc.y), dev.virtbase);

             break;
        case VGA_LED_WRITE_PAD:
             if (copy_from_user(&px, (int *) arg, sizeof(int)))
                  return -EACCES;
             iowrite32(px, dev.virtbase+4);
        //   printk("write pad..\n");
             break;
        case VGA_LED_WRITE_SCORE:
             if (copy_from_user(&score, (int *) arg, sizeof(int)))
                  return -EACCES;
             iowrite32(score, dev.virtbase+8);
```

```
	//	printk("write pad..\n");
		break;
	case VGA_LED_WRITE_LIFE:
		if (copy_from_user(&life, (int *) arg, sizeof(int)))
			return -EACCES;
		iowrite32(life, dev.virtbase+12);
	//	printk("write pad..\n");
		break;
	case VGA_LED_WRITE_ON:
		if(copy_from_user(&on,(int *)arg, sizeof(int)))
			return -EACCES;
		iowrite32(on,dev.virtbase+16);
		break;
	case VGA_LED_WRITE_AUDIO:
		if(copy_from_user(&audio,(int *)arg, sizeof(int)))
			return -EACCES;
		iowrite32(audio,dev.virtbase+20);
		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) and display
 * a welcome message
 */
```

```
static int __init vga_ball_probe(struct platform_device *pdev)
{
	static unsigned char welcome_message[VGA_LED_DIGITS] = {
		0x3E, 0x7D, 0x77, 0x08, 0x38, 0x79, 0x5E, 0x00};
	int i, 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;
	}

	/* Display a welcome message */
	for (i = 0; i < VGA_LED_DIGITS; i++)
		write_digit(i, welcome_message[i]);

	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("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment LED Emulator");
```

lab3-sw/vga_led.h

```
#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>

#define VGA_LED_DIGITS 8

typedef struct {
  unsigned char digit;    /* 0, 1, .. , VGA_LED_DIGITS - 1 */
  unsigned char segments; /* LSB is segment a, MSB is decimal point
*/
} vga_ball_arg_t;
typedef struct {
unsigned int x;
unsigned int y;
} circle_center;



#define VGA_LED_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_LED_WRITE_DIGIT _IOW(VGA_LED_MAGIC, 1, vga_ball_arg_t *)
#define VGA_LED_READ_DIGIT  _IOWR(VGA_LED_MAGIC, 2, vga_ball_arg_t *)
#define VGA_LED_WRITE_CENTER _IOW(VGA_LED_MAGIC, 3, vga_ball_arg_t *)
#define VGA_LED_WRITE_PAD _IOW(VGA_LED_MAGIC, 4, vga_ball_arg_t *)
#define VGA_LED_WRITE_SCORE _IOW(VGA_LED_MAGIC, 5, vga_ball_arg_t *)
#define VGA_LED_WRITE_LIFE _IOW(VGA_LED_MAGIC, 6, vga_ball_arg_t *)
#define VGA_LED_WRITE_ON _IOW(VGA_LED_MAGIC, 7, vga_ball_arg_t *)
#define VGA_LED_WRITE_AUDIO _IOW(VGA_LED_MAGIC, 8, vga_ball_arg_t *)
#endif
```

lab3-sw/wiicontroller.c

```
/**@file wiicontroller.c
 * @brief implementations of the functions communicating with the
wiimote
 */

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


#include "configuration.h"
//#include "wiimote.h"
//#include "wiimote_api.h"
#include "wiicontroller.h"

wiimote_t wii_connect()
{
    wiimote_t wiimote = WIIMOTE_INIT;

    // the address of the wiimote is fixed here
    char *bdaddr = "0C:FC:83:94:9F:99";

    printf("Waiting for connection. Press 1+2 to connect...\n");

    if (wiimote_connect(&wiimote, bdaddr) < 0) {
        fprintf(stderr, "unable to open wiimote: %s\n",
wiimote_get_error());
        exit(1);
    }

    printf("Successfully Connected!\n");

    // turn on the leftmost led
    wiimote.led.one  = 1;

    wiimote.mode.acc = 1;

    // enable the infrared sensor
    wiimote.mode.ir = 1;
    return wiimote;
}

//void wii_getpos(paddle *p){
//}
void wii_getpos(wiimote_t *pwiimote, unsigned int *x, unsigned int
*y)
{
    // x: (0, 1784) -> (0, 512)
    if (wiimote_update(pwiimote) < 0) {
        wiimote_disconnect(pwiimote);
    }
    printf("XXXX  %d\n", pwiimote->ir1.x);
    double scale_factor = 512.0 / CANVAS_SIZE_X;

    // unsigned int x_pos = (pwiimote->ir1.x) * scale_factor;
    unsigned int x_pos = (pwiimote->ir1.x);

    // *x = x_pos >=0 && x_pos <= 512 ? x_pos : NOT_VALID;
    *x = x_pos * 2.5;

    // *x=640-*x;
}

/*
void wii_getpos(wiimote_t *pwiimote, unsigned int *x, unsigned int
*y)
{
    unsigned int x_left_cut = (CAMERA_X_MAX - CAMERA_X)/2;

    unsigned int y_low_cut = (CAMERA_Y_MAX - CAMERA_Y)/2;

    float scale_factor = (float)CANVAS_SIZE_X / (float)CAMERA_X;

    if (wiimote_update(pwiimote) < 0) {
        wiimote_disconnect(pwiimote);
    }

    // project the cooridinates from the wiimote screen to the game
screen
    unsigned int x_pos = (pwiimote->ir1.x - x_left_cut) *
scale_factor;
```

```
    unsigned int y_pos = (pwiimote->ir1.y - y_low_cut) *
scale_factor;

    *x = x_pos >=0 && x_pos <= CANVAS_SIZE_X ? x_pos : NOT_VALID;
    *y = y_pos >=0 && y_pos <= CANVAS_SIZE_Y ? y_pos : NOT_VALID;

    *x=640-*x;
}
*/

void wii_disconnect(wiimote_t *pwiimote){

    wiimote_disconnect(pwiimote);
}
```

lab3-sw/wiicontroller.h

```
/**@file wiicontroller.h
 * @brief the header to the
 */

#include "wiimote.h"
#include "wiimote_api.h"

/**@brief initialize the connection with the wiimote
 *
 * @return the handle to the wiimote
 */
wiimote_t wii_connect();

/**@brief get the current position of the wiimote
 *
 * this function need to be called periodically to keep the wiimote
connected
 */
void wii_getpos(wiimote_t *, unsigned int *, unsigned int *);

/**@brief disconnect the wiimote
 */
void wii_disconnect(wiimote_t *);
```

```
typedef struct {
     unsigned int x;
     unsigned int y;
     wiimote_t w;
} paddle_coords;
```

---

lab3-sw/configuration.h

```
/**@file configuration.h
 * @brief the global configuration for the game
 */

#ifndef CONFIGURATION_H_
#define CONFIGURATION_H_

//! the resolution of the game screen
#define CANVAS_SIZE_X 512
#define CANVAS_SIZE_Y 480

//! so that a free dropping object shows in the screen for 3 secs
#define GRAVITY 0.03

//! the length of a game (in seconds)
#define GAMETIME 60

//! the target score to win a game
#define TARGET 100

//! the maximum number of concurrent sprites allowed at a same time
#define MAX_CONCURRENT_SPRITE 5

//! the maximum distance the ninja can move an each cycle, to make
the sprite more stable
#define MAX_DIFF 10

//! the minimum distance to claim an intersection
#define INTERCTION_THRESHOLD 1000
```

```
//! when to claim the missing of an sprite
#define LOWER_THRESHOLD 80

//! number of different game levels
#define LEVELS 3

//! the invalid valid of coordinates
#define NOT_VALID 9999

//! different type of the objects
typedef enum {HOMEWORK, QUIZ, PROJECT, BOMB, PIZZA} sprite_type;

//! the current screen to display
typedef enum {SELECTION, PLAY, RESULT} screen;

//! the difficulty level
typedef enum {EASY, MEDIUM, HARD} difficulty_level;

//! the range of coordinates reported by wiimote
static const unsigned int CAMERA_X_MAX = 1784;
static const unsigned int CAMERA_Y_MAX = 1272;

// the range of coordinates after doing the scaling
static const unsigned int CAMERA_X = 1696;
static const unsigned int CAMERA_Y = 1272; // 4 x 3 ratio

//! the possible initial speeds for sprits
static const float INIT_VX[] = {0.7, 0.8, 0.9, 1.0, 1.2};
static const float INIT_VY[] = {1.4, 1.45, 1.6, 1.5, 1.55};

//! the possibility of generating new sprite for each type of sprites
static const double POSSIBILITY_MUL = 0.1;
static const float POSSIBILITY_SPRITES[] = {0.4, 0.1, 0.05, 0.01,
0.01};

//! the MULTIPLIER to be applied on possibility and speed to control
the difficulty level
extern float MULTIPLIER;

//! the value of multiple for each difficulty level
static const float MULTIPLIERS[] = {1.0, 1.5, 2.0};

//! the score of each kind of sprite
```

```
static const int SPRITE_SCORE[] = {1, 2, 3, 0, 4};

//! the position of the difficulty selection buttons
static const int POS_SELECTIONS_X[] = {187, 287, 387};
static const int POS_SELECTIONS_Y[] = {300, 300, 300};

//! the position of the try again button
static const int POS_TRY_AGAIN_X = 481;
static const int POS_TRY_AGAIN_Y = 50;

#endif
```

lab3-sw/Makefile

```
ifneq (${KERNELRELEASE},)

# KERNELRELEASE defined: we are being compiled as part of the Kernel
        obj-m := vga_led.o

else

# We are being compiled as a module: use the Kernel build system
     KERNEL_SOURCE := /usr/src/linux
        PWD := $(shell pwd)

default: module hello

module:
     ${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} modules


CC:=gcc
DEFS:=-D_ENABLE_TILT -D_ENABLE_FORCE
CFLAGS:=-Wall -pipe $(DEFS) -O2
INCLUDES:=-I./src
LIBS:=-L./lib -lcwiimote -lbluetooth -lm -lsndfile -pthread

all: hello.o wiicontroller.o
     $(CC) $(CFLAGS) -o hello $^ $(LIBS) $(INCLUDES)

hello.o: hello.c wiicontroller.h vga_led.h configuration.h
```

```
	$(CC) $(CFLAGS) $(INCLUDES) -c $<

wiicontroller.o: wiicontroller.c configuration.h
	$(CC) $(CFLAGS) $(INCLUDES) -c $<

clean:
	${MAKE} -C ${KERNEL_SOURCE} SUBDIRS=${PWD} clean
	${RM} hello

socfpga.dtb : socfpga.dtb
	dtc -O dtb -o socfpga.dtb socfpga.dts

endif
```

16. Verilog Code

BALL.sv

```
/*
 * Avalon memory-mapped peripheral for the VGA LED Emulator
 *
 * Stephen A. Edwards
 * Columbia University
 */

module BALL(input logic        clk,
                 input logic     reset,
            input logic [31:0]  writedata,
            input logic     write,
            input           chipselect,
            input logic  [2:0] address,

            output logic [7:0] VGA_R, VGA_G, VGA_B,
            output logic    VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
            output logic    VGA_SYNC_n,
                  inout AUD_I2C_SDAT,
                  output AUD_I2C_SCLK,
                  output AUD_XCK,
                  output AUD_MUTE,
                  inout AUD_ADCLRCK,
                  input AUD_ADCDAT,
                  inout  AUD_DACLRCK,
                  output AUD_DACDAT,
                  inout AUD_BCLK,
                  input logic [3:0]SW);


   logic [9:0] y_coords, x_coords,px;
     logic [31:0] on;
     logic [31:0] score, life;
     logic [3:0] LED;
     wire main_clk,audio_clk;

wire [1:0] sample_end;
wire [1:0] sample_req;
wire [15:0] audio_output;
wire [15:0] audio_input;

pll pll (
.refclk (clk),
.rst (reset),
.outclk_0 (audio_clk),
.outclk_1 (main_clk)
);

i2c_av_config av_config (
.clk (main_clk),
.reset (reset),
.i2c_sclk (AUD_I2C_SCLK),
.i2c_sdat (AUD_I2C_SDAT),
.status (LED)
);
assign AUD_XCK = audio_clk;
assign AUD_MUTE = (SW != 4'b1111);
audio_codec ac (
.clk (audio_clk),
.reset (reset),
.sample_end (sample_end),
.sample_req (sample_req),
.audio_output (audio_output),
.audio_input (audio_input),
.channel_sel (2'b10),
.AUD_ADCLRCK (AUD_ADCLRCK),
.AUD_ADCDAT (AUD_ADCDAT),
.AUD_DACLRCK (AUD_DACLRCK),
.AUD_DACDAT (AUD_DACDAT),
.AUD_BCLK (AUD_BCLK)
);

wire [15:0] data_connection;
logic pad=1'b0;

audio_effects ae (
.pad(pad),
.apple(data_connection),
//.apple(audio_data[15:0]),
.clk (audio_clk),
.peng(peng),
.sample_end (sample_end[1]),
.sample_req (sample_req[1]),
.audio_output (audio_output),
.audio_input (audio_input),
.control (SW),
.en(en),
.addr_r(addr_r),
.continue_wr(c)
);
logic [10:0] addr_r;
logic en;
logic [31:0] audio_data;
logic peng=1'b0;
logic c;
logic [31:0]last_on;
logic [4:0] counter, counter2;

RAM_Controller ram_controller (.clk50(main_clk),
.sample_req_c(sample_req[1]),.audio_clk(audio_clk),
.data_in(audio_data[15:0]),
.data_out(data_connection),.ren(en),.address_r(addr_r),.c(c));
```

```
   BALL_Emulator led_emulator(.clk50(main_clk),.*);

   always_ff @(posedge clk)
     if (reset) begin
      x_coords <= 10'd320; //
      y_coords <= 10'd240; //
      on <= 32'b11111111111111111111111111111111;
      px <= 10'd1;
      score <= 32'd0;
      life <= 32'd3;
      audio_data <= 32'd0;



     end else if (chipselect && write) begin
           last_on<=on;
           if(on!=last_on) begin
                           peng <= 1'b1;
                           counter <= 5'd0;
           end else begin

                           counter <= counter + 5'd1;
                      end
           if(counter==5'd20) begin
                      peng <= 1'b0;
                      counter <= 5'd0;
           end


           if(x_coords>=px && x_coords<=(px+128) && (y_coords ==
(480-32-32-16))) begin
                pad<=1'b1;
                counter2 <= 5'd0;
           end else begin
                counter2 <= counter2 + 5'd1;
                end
           if(counter2==5'd20) begin
                pad <= 1'b0;
                counter2 <= 5'd0;
           end
   case (address)

     3'b000: begin
       x_coords <= writedata[19:10];
      y_coords <= writedata[9:0];
          end
    3'b001: px <= writedata[9:0];
      3'b010: score <= writedata;
      3'b011: life <= writedata;
      3'b100: on<=writedata;


      3'b101: audio_data <= writedata;
      default:;

       endcase
     end



endmodule
```

BALL_Emulator.sv

```
/*
 * Seven-segment LED emulator
 *
 * Stephen A. Edwards, Columbia University
 */

module BALL_Emulator(
 input logic        clk50, reset,
 input logic [31:0] on,
 input logic peng,
 input logic [9:0]  x_coords,y_coords,
 input logic [31:0] score, life,
 input logic [9:0] px, //paddle address
 output logic [7:0] VGA_R, VGA_G, VGA_B,
 output logic       VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
VGA_SYNC_n);
```

```
/*
 * 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          |____|
 */
   // Parameters for hcount
   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

   // Parameters for vcount
   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


   logic [10:0]                hcount; // Horizontal counter
                                             // Hcount[10:1]
indicates pixel column (0-639)
   logic                  endOfLine;

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

   assign endOfLine = hcount == HTOTAL - 1;
   // Vertical counter
   logic [9:0]                 vcount;
   logic                   endOfField;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)          vcount <= 0;
     else if (endOfLine)
       if (endOfField)   vcount <= 0;
       else              vcount <= vcount + 10'd 1;

   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   // 101 0010 0000 to 101 1101 1111
   assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] ==
3'b111));
   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) );

   /* VGA_CLK is 25 MHz
    *             __    __    __
    * clk50    __|  |__|  |__|
    *
    *             _____       __
    * hcount[0]__|     |_____|
    */
   assign VGA_CLK = hcount[0]; // 25 MHz clock: pixel latched on
rising edge


     logic inZone1, inZone2, inZone3, inZone4, inZone5, inZone6,
inZone7, inZone8, inZone9, inZone10;
   // inZone11, inZone12;
/*
```

```
   assign inZone1 = ((hcount >> 1) == 10'd320) && (vcount >= 0 &&
vcount < 10'd60) || (((hcount >> 1) < 10'd320) && (vcount < 10'd60)
&& (vcount/(10'd320 - (hcount >> 1)) >= 1) || ((hcount >> 1) <
10'd320) && (vcount >= 10'd60) && (vcount/(10'd320 - (hcount >> 1))
>= 1) && ((vcount - 10'd60)/(10'd320 - (hcount >> 1)) < 1)) ||
(((hcount >> 1) >10'd320) && (vcount < 10'd60) && (vcount/((hcount >>
1) - 10'd320) >= 1) || ((hcount >> 1) > 10'd320) && (vcount >=
10'd60) && (vcount/((hcount >> 1) - 10'd320) >= 1) && ((vcount -
10'd60)/((hcount >> 1) - 10'd320) < 1));

   assign inZone2 = ((hcount >> 1) == 10'd320) && (vcount >= 10'd120
&& vcount < 10'd180) || (((hcount >> 1) < 10'd320) && (vcount >=
10'd120 && vcount < 10'd180) && ((vcount - 10'd120)/(10'd320 -
(hcount >> 1)) >= 1) || ((hcount >> 1) < 10'd320) && (vcount >=
10'd180) && ((vcount - 10'd120)/(10'd320 - (hcount >> 1)) >= 1) &&
((vcount - 10'd180)/(10'd320 - (hcount >> 1)) < 1)) || (((hcount >>
1) >10'd320) && (vcount >= 10'd120 && vcount < 10'd180) && ((vcount -
10'd120)/((hcount >> 1) - 10'd320) >= 1) || ((hcount >> 1) > 10'd320)
&& (vcount >= 10'd180) && ((vcount - 10'd120)/((hcount >> 1) -
10'd320) >= 1) && ((vcount - 10'd180)/((hcount >> 1) - 10'd320) <
1));

   assign inZone3 = ((hcount >> 1) == 10'd320) && (vcount >= 10'd240
&& vcount < 10'd300) || (((hcount >> 1) < 10'd320) && (vcount >=
10'd240 && vcount < 10'd300) && ((vcount - 10'd240)/(10'd320 -
(hcount >> 1)) >= 1) || ((hcount >> 1) < 10'd320) && (vcount >=
10'd300) && ((vcount - 10'd240)/(10'd320 - (hcount >> 1)) >= 1) &&
((vcount - 10'd300)/(10'd320 - (hcount >> 1)) < 1)) || (((hcount >>
1) >10'd320) && (vcount >= 10'd240 && vcount < 10'd300) && ((vcount -
10'd240)/((hcount >> 1) - 10'd320) >= 1) || ((hcount >> 1) > 10'd320)
&& (vcount >= 10'd300) && ((vcount - 10'd240)/((hcount >> 1) -
10'd320) >= 1) && ((vcount - 10'd300)/((hcount >> 1) - 10'd320) <
1));

   assign inZone4 = ((hcount >> 1) == 10'd320) && (vcount >= 10'd360
&& vcount < 10'd420) || (((hcount >> 1) < 10'd320) && (vcount >=
10'd360 && vcount < 10'd420) && ((vcount - 10'd360)/(10'd320 -
(hcount >> 1)) >= 1) || ((hcount >> 1) < 10'd320) && (vcount >=
10'd420) && ((vcount - 10'd360)/(10'd320 - (hcount >> 1)) >= 1) &&
((vcount - 10'd420)/(10'd320 - (hcount >> 1)) < 1)) || (((hcount >>
1) >10'd320) && (vcount >= 10'd360 && vcount < 10'd420) && ((vcount -
10'd360)/((hcount >> 1) - 10'd320) >= 1) || ((hcount >> 1) > 10'd320)
&& (vcount >= 10'd420) && ((vcount - 10'd360)/((hcount >> 1) -
10'd320) >= 1) && ((vcount - 10'd420)/((hcount >> 1) - 10'd320) <
1));

   assign inZone5 = (vcount == 0) && ((hcount >> 1) > 10'd200 &&
(hcount >> 1) <= 10'd260) || ((hcount >> 1) >= 10'd200 && (hcount >>
1) < 10'd260) && (2 * vcount/(10'd260 - (hcount >> 1)) <= 1) ||
((hcount >> 1) < 10'd200) && (2 * vcount/(10'd200 - (hcount >> 1)) >
1) && (2 * vcount/(10'd260 - (hcount >> 1)) <= 1);

   assign inZone6 = (vcount == 0) && ((hcount >> 1) > 10'd80 &&
(hcount >> 1) <= 10'd140) || ((hcount >> 1) >= 10'd80 && (hcount >>
1) < 10'd140) && (2 * vcount/(10'd140 - (hcount >> 1)) <= 1) ||
((hcount >> 1) < 10'd80) && (2 * vcount/(10'd80 - (hcount >> 1)) > 1)
&& (2 * vcount/(10'd140 - (hcount >> 1)) <= 1);

   assign inZone7 = (vcount == 0) && ((hcount >> 1) == 10'd20) ||
((hcount >> 1) < 10'd20) && (2 * vcount/(10'd20 - (hcount >> 1)) <=
1);

   assign inZone8 = (vcount == 0) && ((hcount >> 1) < 10'd440 &&
(hcount >> 1) >= 10'd380) || ((hcount >> 1) <= 10'd440 && (hcount >>
1) > 10'd380) && (2 * vcount/((hcount >> 1) - 10'd380) <= 1) ||
((hcount >> 1) > 10'd440) && (2 * vcount/((hcount >> 1) - 10'd440) >
1) && (2 * vcount/((hcount >> 1) - 10'd380) <= 1);

   assign inZone9 = (vcount == 0) && ((hcount >> 1) < 10'd560 &&
(hcount >> 1) >= 10'd500) || ((hcount >> 1) <= 10'd560 && (hcount >>
1) > 10'd500) && (2 * vcount/((hcount >> 1) - 10'd500) <= 1) ||
((hcount >> 1) > 10'd560) && (2 * vcount/((hcount >> 1) - 10'd560) >
1) && (2 * vcount/((hcount >> 1) - 10'd500) <= 1);

   assign inZone10 = (vcount == 0) && ((hcount >> 1) == 10'd620) ||
((hcount >> 1) > 10'd620) && (2 * vcount/((hcount >> 1) - 10'd620) <=
1);
 */
/*
 logic z1_1, z1_2, z1_3, z1_4, z1_5, z2_1, z2_2, z2_3, z2_4, z2_5,
z3_1, z3_2, z3_3, z3_4, z3_5, z4_1, z4_2, z4_3, z4_4, z4_5, z5_1,
z5_2, z5_3, z6_1, z6_2, z6_3, z7_1, z7_2, z8_1, z8_2, z8_3, z9_1,
z9_2, z9_3, z10_1, z10_2;

   assign z1_1 = ((hcount[10:1]) == 10'd320) && (vcount >= 0 &&
vcount < 10'd60);
```

```
   assign z1_2 = ((hcount[10:1]) < 10'd320) && (vcount < 10'd60) &&
(vcount/(10'd320 - (hcount[10:1])) >= 1);
   assign z1_3 = ((hcount[10:1]) < 10'd320) && (vcount >= 10'd60) &&
(vcount/(10'd320 - (hcount[10:1])) >= 1) && ((vcount -
10'd60)/(10'd320 - (hcount[10:1])) < 1);
   assign z1_4 = ((hcount[10:1]) >10'd320) && (vcount < 10'd60) &&
(vcount/((hcount[10:1]) - 10'd320) >= 1);
   assign z1_5 = ((hcount[10:1]) > 10'd320) && (vcount >= 10'd60) &&
(vcount/((hcount[10:1]) - 10'd320) >= 1) && ((vcount -
10'd60)/((hcount[10:1]) - 10'd320) < 1);
   assign inZone1 = z1_1 || z1_2 || z1_3 || z1_4 || z1_5;

   assign z2_1 = ((hcount[10:1]) == 10'd320) && (vcount >= 10'd120 &&
vcount < 10'd180);
   assign z2_2 = ((hcount[10:1]) < 10'd320) && (vcount >= 10'd120 &&
vcount < 10'd180) && ((vcount - 10'd120)/(10'd320 - (hcount[10:1]))
>= 1);
   assign z2_3 = ((hcount[10:1]) < 10'd320) && (vcount >= 10'd180) &&
((vcount - 10'd120)/(10'd320 - (hcount[10:1])) >= 1) && ((vcount -
10'd180)/(10'd320 - (hcount[10:1])) < 1);
   assign z2_4 = ((hcount[10:1]) >10'd320) && (vcount >= 10'd120 &&
vcount < 10'd180) && ((vcount - 10'd120)/((hcount[10:1]) - 10'd320)
>= 1);
   assign z2_5 = ((hcount[10:1]) > 10'd320) && (vcount >= 10'd180) &&
((vcount - 10'd120)/((hcount[10:1]) - 10'd320) >= 1) && ((vcount -
10'd180)/((hcount[10:1]) - 10'd320) < 1);
   assign inZone2 = z2_1 || z2_2 || z2_3 || z2_4 || z2_5;

   assign z3_1 = ((hcount[10:1]) == 10'd320) && (vcount >= 10'd240 &&
vcount < 10'd300);
   assign z3_2 = ((hcount[10:1]) < 10'd320) && (vcount >= 10'd240 &&
vcount < 10'd300) && ((vcount - 10'd240)/(10'd320 - (hcount[10:1]))
>= 1);
   assign z3_3 = ((hcount[10:1]) < 10'd320) && (vcount >= 10'd300) &&
((vcount - 10'd240)/(10'd320 - (hcount[10:1])) >= 1) && ((vcount -
10'd300)/(10'd320 - (hcount[10:1])) < 1);
   assign z3_4 = ((hcount[10:1]) >10'd320) && (vcount >= 10'd240 &&
vcount < 10'd300) && ((vcount - 10'd240)/((hcount[10:1]) - 10'd320)
>= 1);
   assign z3_5 = ((hcount[10:1]) > 10'd320) && (vcount >= 10'd300) &&
((vcount - 10'd240)/((hcount[10:1]) - 10'd320) >= 1) && ((vcount -
10'd300)/((hcount[10:1]) - 10'd320) < 1);
   assign inZone3 = z3_1 || z3_2 || z3_3 || z3_4 || z3_5;
   assign z4_1 = ((hcount[10:1]) == 10'd320) && (vcount >= 10'd360 &&
vcount < 10'd420);
   assign z4_2 = (((hcount[10:1]) < 10'd320) && (vcount >= 10'd360 &&
vcount < 10'd420) && ((vcount - 10'd360)/(10'd320 - (hcount[10:1]))
>= 1));
   assign z4_3 = ((hcount[10:1]) < 10'd320) && (vcount >= 10'd420) &&
((vcount - 10'd360)/(10'd320 - (hcount[10:1])) >= 1) && ((vcount -
10'd420)/(10'd320 - (hcount[10:1])) < 1);
   assign z4_4 = ((hcount[10:1]) >10'd320) && (vcount >= 10'd360 &&
vcount < 10'd420) && ((vcount - 10'd360)/((hcount[10:1]) - 10'd320)
>= 1);
   assign z4_5 = ((hcount[10:1]) > 10'd320) && (vcount >= 10'd420) &&
((vcount - 10'd360)/((hcount[10:1]) - 10'd320) >= 1) && ((vcount -
10'd420)/((hcount[10:1]) - 10'd320) < 1);
   assign inZone4 = z4_1 || z4_2 || z4_3 || z4_4 || z4_5;

   assign z5_1 = (vcount == 0) && ((hcount[10:1]) > 10'd200 &&
(hcount[10:1]) <= 10'd260);
   assign z5_2 = ((hcount[10:1]) >= 10'd200 && (hcount[10:1]) <
10'd260) && (2 * vcount/(10'd260 - (hcount[10:1])) <= 1);
   assign z5_3 = ((hcount[10:1]) < 10'd200) && (2 * vcount/(10'd200 -
(hcount[10:1])) > 1) && (2 * vcount/(10'd260 - (hcount[10:1])) <= 1);
   assign inZone5 = z5_1 || z5_2 || z5_3;

   assign z6_1 = (vcount == 0) && ((hcount[10:1]) > 10'd80 &&
(hcount[10:1]) <= 10'd140);
   assign z6_2 = ((hcount[10:1]) >= 10'd80 && (hcount[10:1]) <
10'd140) && (2 * vcount/(10'd140 - (hcount[10:1])) <= 1);
   assign z6_3 = ((hcount[10:1]) < 10'd80) && (2 * vcount/(10'd80 -
(hcount[10:1])) > 1) && (2 * vcount/(10'd140 - (hcount[10:1])) <= 1);
   assign inZone6 = z6_1 || z6_2 || z6_3;

   assign z7_1 = (vcount == 0) && ((hcount[10:1]) == 10'd20);
   assign z7_2 = ((hcount[10:1]) < 10'd20) && (2 * vcount/(10'd20 -
(hcount[10:1])) <= 1);
   assign inZone7 = z7_1 || z7_2;

   assign z8_1 = (vcount == 0) && ((hcount[10:1]) < 10'd440 &&
(hcount[10:1]) >= 10'd380);
   assign z8_2 = ((hcount[10:1]) <= 10'd440 && (hcount[10:1]) >
10'd380) && (2 * vcount/((hcount[10:1]) - 10'd380) <= 1);
```

```
   assign z8_3 = ((hcount[10:1]) > 10'd440) && (2 *
vcount/((hcount[10:1]) - 10'd440) > 1) && (2 * vcount/((hcount[10:1])
- 10'd380) <= 1);
   assign inZone8 = z8_1 || z8_2 || z8_3;

   assign z9_1 = (vcount == 0) && ((hcount[10:1]) < 10'd560 &&
(hcount[10:1]) >= 10'd500);
   assign z9_2 = ((hcount[10:1]) <= 10'd560 && (hcount[10:1]) >
10'd500) && (2 * vcount/((hcount[10:1]) - 10'd500) <= 1);
   assign z9_3 = ((hcount[10:1]) > 10'd560) && (2 *
vcount/((hcount[10:1]) - 10'd560) > 1) && (2 * vcount/((hcount[10:1])
- 10'd500) <= 1);
   assign inZone9 = z9_1 || z9_2 || z9_3;

   assign z10_1 = (vcount == 0) && ((hcount[10:1]) == 10'd620);
   assign z10_2 = ((hcount[10:1]) > 10'd620) && (2 *
vcount/((hcount[10:1]) - 10'd620) <= 1);
   assign inZone10 = z10_1 || z10_2;

*/

logic[26:0]    changeColor;

   always_ff @(posedge clk50 or posedge reset)
     if (reset)  changeColor <= 0;
     else if (changeColor[25:0] == 26'h3ffffff)  begin
        changeColor[26] <= 1 - changeColor[26];
        changeColor[25:0] <= 0;
     end
     else   changeColor <= changeColor + 27'd 1;


logic[6:0] line;
//always_ff @(posedge clk50) begin
//
//     if(vcount[6:0] > 7'd119) begin
//         line = vcount[6:0] - 7'd120;
//     end
//     else
//         line = vcount[6:0];
//end

assign line = vcount%120;
```

```
logic[19:0] left_sprite_data;
logic background;

left_rom left_unit(
      .clock(clk50),
      .address(line),
      .q(left_sprite_data)
);


logic[19:0] right_sprite_data;

right_rom right_unit(
      .clock(clk50),
      .address(line),
      .q(right_sprite_data)
);

logic[59:0] mid_sprite_data;
logic[1:0] mid_sprite;
logic[8:0] mid_address;

mid_rom mid_unit(
      .clock(clk50),
      .address(mid_address),
      .q(mid_sprite_data)
);



   logic isBall; // hcount or vcount are close to the center of the
ball

   logic [10:0]                        x_dist;
   logic [10:0]                        y_dist;

      //
      logic  [5:0]                        character_address;
      logic                                        rom_output;
      logic [7:0]                                  char_rom_data;
      /* x distance has to be divided by two since x_axis has double
the pixels for some reason */
      assign x_dist = (x_coords > hcount[10:1]) ? ((x_coords -
hcount[10:1])):((hcount[10:1] - x_coords));
      assign y_dist = (y_coords > vcount) ? (y_coords -
vcount):(vcount - y_coords);

   assign isBall = ( (x_dist*x_dist + y_dist*y_dist) < 200);


      //
      char_gen_rom char_rom_unit(
                 .clock(clk50),
                 .address({character_address,vcount[2:0]}),
                 .q(char_rom_data)
      );

      logic [23:0] brick_output;
      logic [14:0] brick_address;
      logic [3:0]  brick_type;//100.011.010,001,000 brick_type=4'o3
{brick_type, brick_type*32+hcount[7:1]}

      bricks brick_unit(
      .clock(clk50),
      .address(brick_address),
      .q(brick_output)
      );
      logic en;

      always_ff @(posedge clk50) begin

           if(on[0] && hcount[10:8] == 3'd0 && vcount[8:5] == 4'd1 )
begin
                 en <= 1'b1;

brick_address<=hcount[10:8]*32*128+vcount[4:0]*128+hcount[7:1];
           end
           else if(on[1] && hcount[10:8] == 3'd1 && vcount[8:5] ==
4'd1 ) begin
                 en <= 1'b1;

brick_address<=hcount[10:8]*32*128+vcount[4:0]*128+hcount[7:1];
                 end
```

```
          else if(on[2] && hcount[10:8] == 3'd2 && vcount[8:5] ==
4'd1 ) begin
                en <= 1'b1;

brick_address<=hcount[10:8]*32*128+vcount[4:0]*128+hcount[7:1];
                end
          else if(on[3] && hcount[10:8] == 3'd3 && vcount[8:5] ==
4'd1) begin
                en <= 1'b1;

brick_address<=(hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1];
          end
          else if(on[4] && hcount[10:8] == 3'd4 && vcount[8:5] ==
4'd1 ) begin
                en <= 1'b1;

brick_address<=hcount[10:8]*32*128+vcount[4:0]*128+hcount[7:1];
                end

          else if(on[5] && hcount[10:8] == 3'd0 && vcount[8:5] ==
4'd2 ) begin
                en <= 1'b1;
                brick_address<=
(1'b1+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1];
          end
          else if(on[6] && hcount[10:8] == 3'd1 && vcount[8:5] ==
4'd2 ) begin
                en <= 1'b1;

brick_address<=(1'b1+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1]
;
                end
          else if(on[7] && hcount[10:8] == 3'd2 && vcount[8:5] ==
4'd2 ) begin
                en <= 1'b1;

brick_address<=(1'b1+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1]
;
                end
          else if(on[8] && hcount[10:8] == 3'd3 && vcount[8:5] ==
4'd2) begin
                en <= 1'b1;
```

```
brick_address<=(hcount[10:8]+1'b1)*32*128+vcount[4:0]*128+hcount[7:1]
;
          end


          else if(on[10] && hcount[10:8] == 3'd0 && vcount[8:5] ==
4'd3 ) begin
               en <= 1'b1;

brick_address<=(2'd2+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1]
;
          end
          else if(on[11] && hcount[10:8] == 3'd1 && vcount[8:5] ==
4'd3 ) begin
               en <= 1'b1;

brick_address<=(2'd2+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1]
;
               end
          else if(on[13] && hcount[10:8] == 3'd3 && vcount[8:5] ==
4'd3) begin
               en <= 1'b1;

brick_address<=(hcount[10:8]+2'd2)*32*128+vcount[4:0]*128+hcount[7:1]
;
          end
               else if(on[14] && hcount[10:8] == 3'd4 && vcount[8:5]
== 4'd3) begin
               en <= 1'b1;
               brick_address<= (hcount[10:8]-3'd4)
*32*128+vcount[4:0]*128+hcount[7:1];
          end

          else if(on[15] && hcount[10:8] == 3'd0 && vcount[8:5] ==
4'd4 ) begin
               en <= 1'b1;

brick_address<=(2'd3+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1]
;
          end
          else if(on[16] && hcount[10:8] == 3'd1 && vcount[8:5] ==
4'd4 ) begin
```

```
                en <= 1'b1;

brick_address<=(2'd3+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1]
;
                end
           else if(on[17] && hcount[10:8] == 3'd2 && vcount[8:5] ==
4'd4 ) begin
                en <= 1'b1;

brick_address<=(2'd3+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1]
;
                end
           else if(on[19] && hcount[10:8] == 3'd4 && vcount[8:5] ==
4'd4) begin
                en <= 1'b1;
                brick_address<= (hcount[10:8]-2'd3)
*32*128+vcount[4:0]*128+hcount[7:1];
           end

           else if(on[20] && hcount[10:8] == 3'd0 && vcount[8:5] ==
4'd5 ) begin
                en <= 1'b1;

brick_address<=(3'd3+hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1]
;
           end
           else if(on[21] && hcount[10:8] == 3'd1 && vcount[8:5] ==
4'd5 ) begin
                en <= 1'b1;

brick_address<=(hcount[10:8])*32*128+vcount[4:0]*128+hcount[7:1];
                end
           else if(on[22] && hcount[10:8] == 3'd2 && vcount[8:5] ==
4'd5 ) begin
                en <= 1'b1;

brick_address<=(hcount[10:8]-3'd2)*32*128+vcount[4:0]*128+hcount[7:1]
;
                end
           else if(on[23] && hcount[10:8] == 3'd3 && vcount[8:5] ==
4'd5) begin
                en <= 1'b1;
                brick_address<= (hcount[10:8]+3'd1)
*32*128+vcount[4:0]*128+hcount[7:1];
          end
                else if(on[24] && hcount[10:8] == 3'd4 && vcount[8:5]
== 4'd5) begin
                en <= 1'b1;
                brick_address<= (hcount[10:8]-3'd2)
*32*128+vcount[4:0]*128+hcount[7:1];
          end

          else begin
                en <= 1'b0;
          end

     end

     logic [23:0] paddle_output;
     logic  [11:0] paddle_address;
     logic paddle_en;

     paddle paddle_unit(
     .clock(clk50),
     .address(paddle_address),
     .q(paddle_output)
     );


     always_ff @(posedge clk50) begin

          if(vcount[8:5]==4'd13 && (hcount[10:1]-px)<=127 &&
(hcount[10:1]-px>=0)) begin
                paddle_en<=1'b1;
                paddle_address<= vcount[4:0]*128+
(hcount[10:1]-px)%128; //hcount[7:1];
          end
          else if(on[9] && hcount[10:8] == 3'd4 && vcount[8:5] ==
4'd2) begin
                paddle_en<=1'b1;
                paddle_address<= vcount[4:0]*128+
(hcount[10:1]-px)%128; //hcount[7:1];
          end
          else if(on[18] && hcount[10:8] == 3'd3 && vcount[8:5] ==
4'd4) begin
```

```
                paddle_en<=1'b1;
                paddle_address<= vcount[4:0]*128+
(hcount[10:1]-px)%128; //hcount[7:1];
           end
           else if(on[12] && hcount[10:8] == 3'd2 && vcount[8:5] ==
4'd3 ) begin
                paddle_en<=1'b1;
                paddle_address<= vcount[4:0]*128+
(hcount[10:1]-px)%128; //hcount[7:1];
           end
           else paddle_en<=1'b0;

      end


      logic [23:0] ball_output;
      logic  [9:0] ball_address;
      logic ball_en;
      logic [4:0] ball_pixel_y;
      //logic [4:0] ball_pixel_y;
      logic [5:0] ball_pixel_x;
      ball ball_unit(
           .clock(clk50),
           .address(ball_address),
           .q(ball_output)
      );



logic ball_reset;
assign ball_reset=(hcount[10:1]==10'd0)&&(vcount==10'd0);
      always_ff @(posedge clk50 or posedge ball_reset) begin
           if(ball_reset) begin
                ball_en<=1'b0;
                ball_pixel_y<=5'd0;
                ball_pixel_x<=6'd0;
           end
           else if ((vcount<=(y_coords+10'd16) &&
vcount>(y_coords-10'd16))  && (hcount[10:1]<=(x_coords+10'd16) &&
hcount[10:1]>(x_coords-10'd16) )) begin
                ball_en<=1'b1;
                ball_address<= ball_pixel_y*32+ball_pixel_x[5:1];
                if(ball_pixel_x==6'b111111) begin
                     ball_pixel_y<=ball_pixel_y+1'b1;
                     ball_pixel_x<=6'd0;
                end
                else
                     ball_pixel_x<=ball_pixel_x+1'b1;
           end
           else begin
                ball_en<=1'b0;

           end
      end

      assign rom_output=char_rom_data[~hcount[3:1]];//hcount LSB =>
char_rom_data MSB

      always_comb begin
           character_address=6'o40;// space by default
           if (vcount<8) begin
                case (hcount[10:4])
                     7'd1: character_address=6'o14;
                     7'd2: character_address=6'o11;
                     7'd3: character_address=6'o26;
                     7'd4: character_address=6'o05;
                     7'd5: character_address=6'o23;

                     7'd7: begin
                           case(life)
                           32'd0: character_address=6'o60; //ten
                           32'd1: character_address=6'o61; //ten
                           32'd2: character_address=6'o62; //ten
                           32'd3: character_address=6'o63; //ten
                           endcase
                           end
//                   7'd77: character_address=6'd48+(score/100);
//hundred
                     7'd77: begin
                           case(score/10)
                           32'd0: character_address=6'o60; //ten
                           32'd1: character_address=6'o61; //ten
                           32'd2: character_address=6'o62; //ten
                           32'd3: character_address=6'o63; //ten
                           32'd4: character_address=6'o64; //ten
```

```
                                32'd5: character_address=6'o65; //ten
                                32'd6: character_address=6'o66; //ten
                                32'd7: character_address=6'o67; //ten
                                32'd8: character_address=6'o70; //ten
                                32'd9: character_address=6'o71; //ten
                                endcase
                                end
                        7'd78: begin
                                case(score%10)
                                32'd0: character_address=6'o60; //ten
                                32'd1: character_address=6'o61; //ten
                                32'd2: character_address=6'o62; //ten
                                32'd3: character_address=6'o63; //ten
                                32'd4: character_address=6'o64; //ten
                                32'd5: character_address=6'o65; //ten
                                32'd6: character_address=6'o66; //ten
                                32'd7: character_address=6'o67; //ten
                                32'd8: character_address=6'o70; //ten
                                32'd9: character_address=6'o71; //ten
                                endcase
                                end
                        7'd71: character_address=6'o23; //s
                        7'd72: character_address=6'o03; //c
                        7'd73: character_address=6'o17; //o
                        7'd74: character_address=6'o22; //r
                        7'd75: character_address=6'o05; //e

                    endcase


                end

            end

    /*
      always_comb begin
         {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00}; // Black
               if(en)
                     {VGA_R, VGA_G, VGA_B} = brick_output;
               if(paddle_en)
                     {VGA_R, VGA_G, VGA_B} = paddle_output;
               if(ball_en)
                     {VGA_R, VGA_G, VGA_B} = ball_output;
```

```
          //if (isBall)
               //{VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};
          if (rom_output)
               {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'h00};
  end
*/

always_comb begin

    if (hcount[10:1] > 19 && hcount[10:1] < 80 || hcount[10:1] > 139
&& hcount[10:1] < 200 || hcount[10:1] > 259 && hcount[10:1] < 320 )
         mid_sprite = 0;
    else if (hcount[10:1] > 79 && hcount[10:1] < 140 || hcount[10:1]
> 199 && hcount[10:1] < 260)
         mid_sprite = 1;
    else if(hcount[10:1] > 319 && hcount[10:1] < 380 || hcount[10:1]
> 439 && hcount[10:1] < 500 || hcount[10:1] > 559 && hcount[10:1] <
620 )
         mid_sprite = 2;
    else
         mid_sprite = 3;
end

assign mid_address = mid_sprite * 120 + line;

always_comb begin
    if (hcount[10:1] < 20)
       background = left_sprite_data[19 - hcount[10:1]];
    else if (hcount[10:1] > 619)
       background = right_sprite_data[639 - hcount[10:1]];
//    else if (hcount[10:1] > 19 && hcount[10:1] < 80)
//       background = mid_sprite_data[79 - hcount[10:1]];
//    else if (hcount[10:1] > 79 && hcount[10:1] < 140)
//       background = mid_sprite_data[139 - hcount[10:1]];
//    else if (hcount[10:1] > 139 && hcount[10:1] < 200)
//       background = mid_sprite_data[199 - hcount[10:1]];
//    else if (hcount[10:1] > 199 && hcount[10:1] < 260)
//       background = mid_sprite_data[259 - hcount[10:1]];
//    else if (hcount[10:1] > 259 && hcount[10:1] < 320)
//       background = mid_sprite_data[319 - hcount[10:1]];
//    else if (hcount[10:1] > 319 && hcount[10:1] < 380)
//       background = mid_sprite_data[379 - hcount[10:1]];
//    else if (hcount[10:1] > 379 && hcount[10:1] < 440)
//        background = mid_sprite_data[439 - hcount[10:1]];
//     else if (hcount[10:1] > 439 && hcount[10:1] < 500)
//        background = mid_sprite_data[499 - hcount[10:1]];
//     else if (hcount[10:1] > 499 && hcount[10:1] < 560)
//        background = mid_sprite_data[559 - hcount[10:1]];
//     else
//        background = mid_sprite_data[619 - hcount[10:1]];
    else
       background = mid_sprite_data[59 - (hcount[10:1] - 21)%60];
end


     always_comb begin
/*
     if(peng)
     {VGA_R, VGA_G, VGA_B} = {8'h00, 8'h00, 8'h00}; // Black
     else
     {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'hff}; // whilte
*/

          if (background && (changeColor[26])) begin
             {VGA_R, VGA_G, VGA_B} = {8'd60, 8'd210, 8'd144}; //
Light Green
      end else if (background && (~changeColor[26]))
           {VGA_R, VGA_G, VGA_B} = {8'd40, 8'd145, 8'd100}; // Dark
Green
        else if ((~background) && (changeColor[26]))
             {VGA_R, VGA_G, VGA_B} = {8'd40, 8'd145, 8'd100}; //
Dark Green
      else //if ((~background) && (~changeColor[26]))
           {VGA_R, VGA_G, VGA_B} = {8'd60, 8'd210, 8'd144}; // Light
Green


      if(en) begin
               {VGA_R, VGA_G, VGA_B} = brick_output;
          end
          if(paddle_en)
               {VGA_R, VGA_G, VGA_B} = paddle_output;
          if(ball_en&&ball_output)
               {VGA_R, VGA_G, VGA_B} = ball_output;
          //if (isBall)
               //{VGA_R, VGA_G, VGA_B} = {8'h00, 8'hff, 8'h00};
          if (rom_output)
               {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'h00};



   end

endmodule // VGA_LED_Emulator
```

---

audio_codec.v

```
module audio_codec (
    input  clk,
    input  reset,
    output [1:0]  sample_end,
    output [1:0]  sample_req,
    input  [15:0] audio_output,
    output [15:0] audio_input,
    // 1 - left, 0 - right
    input  [1:0] channel_sel,

    output AUD_ADCLRCK,
    input AUD_ADCDAT,
    output AUD_DACLRCK,
    output AUD_DACDAT,
    output AUD_BCLK
);

reg [7:0] lrck_divider;
reg [1:0] bclk_divider;

reg [15:0] shift_out;
reg [15:0] shift_temp;
reg [15:0] shift_in;

wire lrck = !lrck_divider[7];

assign AUD_ADCLRCK = lrck;
assign AUD_DACLRCK = lrck;
assign AUD_BCLK = bclk_divider[1];
assign AUD_DACDAT = shift_out[15];

always @(posedge clk) begin
    if (reset) begin
        lrck_divider <= 8'hff;
        bclk_divider <= 2'b11;
    end else begin
        lrck_divider <= lrck_divider + 1'b1;
        bclk_divider <= bclk_divider + 1'b1;
    end
end

assign sample_end[1] = (lrck_divider == 8'h40);
assign sample_end[0] = (lrck_divider == 8'hc0);
assign audio_input = shift_in;
assign sample_req[1] = (lrck_divider == 8'hfe);
assign sample_req[0] = (lrck_divider == 8'h7e);

wire clr_lrck = (lrck_divider == 8'h7f);
wire set_lrck = (lrck_divider == 8'hff);
// high right after bclk is set
wire set_bclk = (bclk_divider == 2'b10 && !lrck_divider[6]);
// high right before bclk is cleared
wire clr_bclk = (bclk_divider == 2'b11 && !lrck_divider[6]);

always @(posedge clk) begin
    if (reset) begin
        shift_out <= 16'h0;
        shift_in <= 16'h0;
        shift_in <= 16'h0;
    end else if (set_lrck || clr_lrck) begin
        // check if current channel is selected
        if (channel_sel[set_lrck]) begin
            shift_out <= audio_output;
            shift_temp <= audio_output;
            shift_in <= 16'h0;
        // repeat the sample from the other channel if not
        end else shift_out <= shift_temp;
    end else if (set_bclk == 1) begin
        // only read in if channel is selected
        if (channel_sel[lrck])
            shift_in <= {shift_in[14:0], AUD_ADCDAT};
    end else if (clr_bclk == 1) begin
```

```
        shift_out <= {shift_out[14:0], 1'b0};
    end
end

endmodule
```

---

audio_effects.v

```
module audio_effects (
input [15:0] apple,
input pad,
input clk,
input peng,
input sample_end,
input sample_req,
output [15:0] audio_output,
input [15:0] audio_input,
input [3:0] control,
output en,
output addr_r,
output continue_wr
);

sound_2 ti_ta(
      .clock(clk),
      .address(back_index),
      .q(audio_back_out)
);

reg continue;
reg [18:0] bg_address;
wire [15:0] bg_output;

DING ding(
      .clock(clk),
      .address(ding_index),
      .q(audio_ding_out)
);

qq QQ(
     .clock(clk),
     .address(qq_index),
     .q(audio_qq_out)
);


//logic [15:0]              mem [15:0];
reg [10:0] address_r;
reg ren;

wire [15:0] audio_sample_out;
wire [15:0] audio_back_out;
reg play=1'd1;
reg play_ding=1'd1;
reg [12:0] audio_index=13'd0;
reg [13:0] qq_index = 14'd0;
reg [13:0] back_index=14'd0;
reg [13:0] ding_index=14'd0;
wire [15:0] audio_ding_out;
wire [15:0] audio_qq_out;
reg [15:0] romdata [0:99];
reg [6:0] index = 7'd0;
reg [15:0] last_sample;
reg [15:0] dat;
assign audio_output = dat;
parameter SINE = 0;
parameter FEEDBACK = 1;
reg ding_ready;

always @(posedge clk) begin
if(peng)
          ding_ready<=1'b1;
if (ding_index==14'd10538)
                ding_ready<=1'b0;

end
reg [31:0] count=0;
     always @(posedge clk) begin

          if (sample_end) begin
               last_sample <= audio_input;
          end
          if (sample_req) begin
          //control==4'b0100 ||
               if (control==4'b0100 ||peng || ding_index>14'b0 )
begin
                    dat<=audio_ding_out;
                    if (ding_index==14'd10538) begin
                         ding_index<=14'd0;


                    end else begin

                         ding_index<=ding_index+1'd1;
                    end
               end else if(pad || back_index>14'd0) begin
                    dat <= audio_back_out;
                    if (back_index == 14'd11519) begin
                         back_index <= 14'd00;
          //             play<=1'd0;
                    end else begin
                         back_index <= back_index + 1'b1;
                         end

               end else begin
     /*
                    dat <= audio_qq_out;
                    if (qq_index == 14'd5139) begin
                         qq_index <= 14'd00;
          //             play<=1'd0;
                    end else begin
                         qq_index <= qq_index + 1'b1;
                         end
     */
               dat <= 16'd10551;
          end

               //        dat <= 16'd0;
     /*
               if(control==4'b0010) begin
                         ren <= 1'b1;
                         dat<=apple;
                         if (address_r<=2046) begin
                              address_r <= address_r +11'd1;
                              continue <= 1'b0;
                         end
```

```
                        else begin
                             continue <= 1'b1;
                             address_r <= 11'd0;
                        end
              end
          */
              /*
              if (control==4'b1000) begin
                   dat <= last_sample;
                   play<=1'd1;
                   play_ding<=1'd1;
              end else if (control[SINE]&&play) begin
                   dat <= audio_back_out;
                   if (back_index == 14'd11519) begin
                        back_index <= 14'd00;
                        play<=1'd0;
                   end else
                        back_index <= back_index + 1'b1;

              end else if(control==4'b0010) begin
                        ren <= 1'b1;
                        dat<=apple;
                        if (address_r<=2046) begin
                             address_r <= address_r +11'd1;
                             continue <= 1'b0;
                        end
                        else begin
                             continue <= 1'b1;
                             address_r <= 11'd0;
                        end
              end
                        */

     end //end of sample request

end
     assign en=ren;
     assign addr_r=address_r;
     assign continue_wr = continue;


endmodule
```