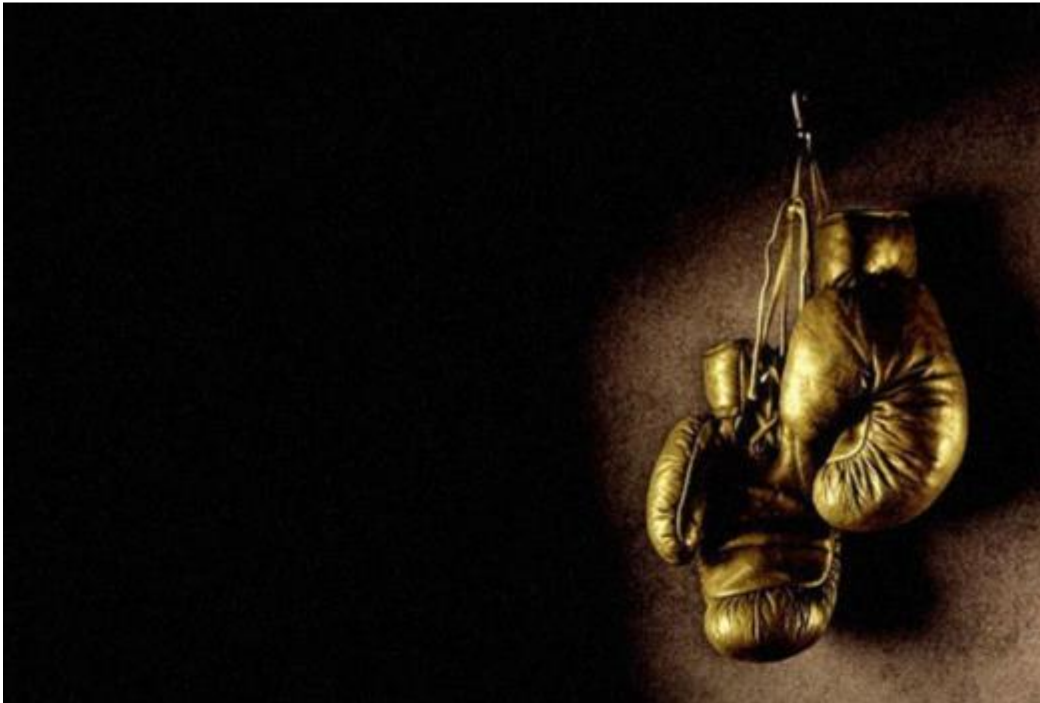
CSEE 4840 Embedded System Project Report

Video Game: Real Boxing

Jiaqi Guo jg3639



Contents:

1. **Introduction** ... 3
2. **VGA Image Display Module** ... 4
   - 2.1 Sprite Graphics ... 4
   - 2.2 Techniques to Save Memory ... 7
3. **SSM2603 Audio Codec Module** ... 9
4. **Hardware Interface** ... 10
5. **Logic Control Unit** ... 13
6. **Device Driver Unit** ... 14
7. **Game Demo** ... 14
8. **Summary and Future Direction** ... 16
   - 8.1 Summary ... 16
   - 8.2 Future Direction ... 16
9. **Appendix** ... 16
   - 9.1 Software ... 16
     - 9.1.1 boxing.c ... 16
     - 9.1.2 usbkeyboard.c ... 29
     - 9.1.3 usbkeyboard.h ... 30
     - 9.1.4 vga_led.c ... 30
     - 9.1.4 vga_led.h ... 35
     - 9.1.5 Makefile ... 36
   - 9.2 Hardware ... 36
     - 9.2.1 VGA_LED.sv ... 36
     - 9.2.2 VGA_LED_Emulator.sv ... 38
     - 9.2.2 Audio_effects.sv ... 61
     - 9.2.3 Audio_codec.sv ... 64
     - 9.2.3 Audio_Top.sv ... 66

1. Introduction

For this project, the goal is to implement a boxing game via FPGA Cyclone board and several peripherals such as VGA monitor, SSM2603 audio codec and USB keyboard. In this game, players could join in a boxing fight versus a computer opponent. For controlling part, players could use LEFT and RIGHT buttons to set up corresponding attack on the opponent, and SPACE or DOWN buttons to defend opponent's attack. Besides, both player and AI opponent have a life bar to denote their current health status, once a player runs out of all health, the game ending condition is met and his opponent wins the game. I also designed a simple user interface for the starting screen. In addition to image display, this game also contains three sound effects for attack, defend and dodge. Figure 1 shows the screen when the game is counting for start.

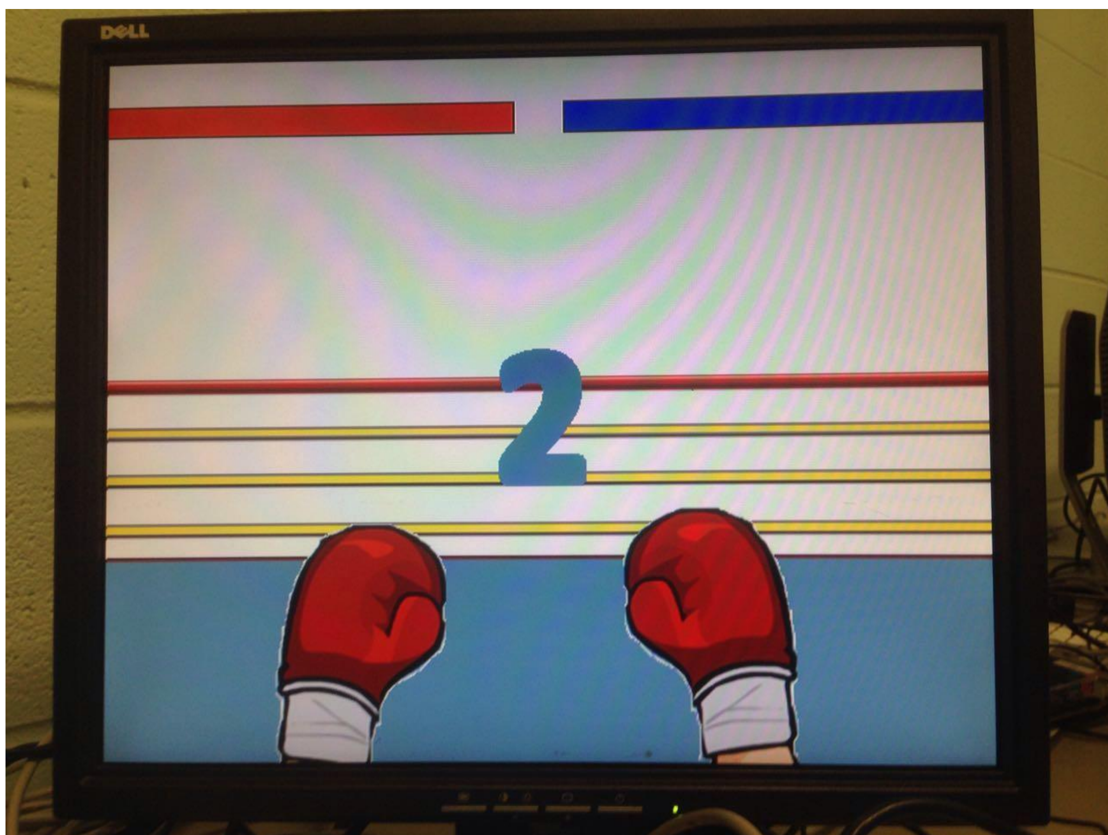


Figure 1. **Project Demo.**

In general, this game combines both hardware and software design, and would be a decent practice for what I have learned in the class. The software section contains device driver unit and logic control unit. The device driver unit provides an interface between hardware (audio codec, keyboard and VGA monitor) data ports and logic control unit. And the logic control unit provides most controlling judgments while the game proceeds. These controlling judgments include game starting judgment, hit or miss judgment, successful defense judgment and game ending judgment.

The hardware section mainly contains the design of VGA image display module and SSM2603 audio codec module. As mention above, the device driver unit provides interface for logic control unit and hardware, in more detail, logic control unit could send specific data to hardware through device drivers. After the peripherals receive data from their drivers, they decode the data and display the corresponding images or music. The overall flowchart can be shown as follows:

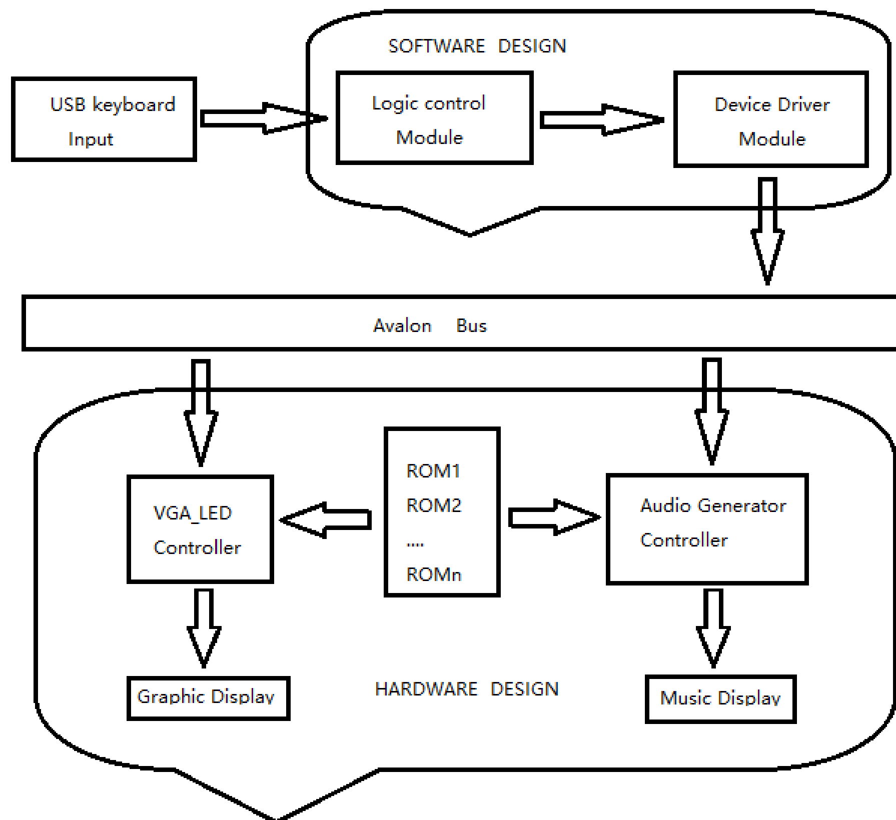


Figure 2. **Project Flowchart.**

2. VGA Image Display Module

I started working on my project from the design of VGA image display module. This part receives data from VGA_LED device driver, decode the data into display commands and print the screen according to these commands. The screen contains 640x480 pixels, and each pixel has 3 channels for RGB information. While displaying, the hardware scrolls down every pixel on the screen in a zigzag order. And we need to decide what RGB value should be for each pixel.

2.1 Sprite Graphics

The game contains several images to display, like player's left and right fists, the opponent, background and life bars. However, these images often get overlapped with each other while displaying. Therefore, we will need several different layers that each

contains one of those images. And what would be finally shown on the screen is determined by the priority of these layers.

The technique to display the images in several different layers and use layer's priority to determine what is shown on the screen is called the Sprite Graphics. At each pixel, the VGA controller first finds out how many sprites have values at this point, and each sprite reads in data from its ROM. Then through a priority controller, VGA module finally decides which sprite to display at this pixel.

In this project, there are 14 sprites and 11 of them need a ROM to store the image information. For those sprites which do not need extra image information, the following two conditions should be met:

(1) The sprite shape could be described in a simple equation, such as a line, a circle or a rectangle.
(2) The sprite should be monochrome.

And in this program, only the life bar and life bar border would meet these conditions. Thus they do not consume extra memory on the Sockit board. However, in some other cases, only the shape information of the sprite is needed, and we could play some tricks to save memory in ROM setting up of these sprites. This will be explored in more detail in the following section.

For those sprites which require shape or color information, we will need a ROM to store the image information for the sprite. In order to build a ROM, we first use Matlab to convert a RGB image into a mif file (the Matlab code to convert RGB image into mif file is added to appendix), and this file would be used to initialize the corresponding ROM. And in QUARTUS II, we could use the MegeWizard function to generate a ROM, and we only need to specify the width of data-output port, the number of memory units and the initializing file.

Some of the sprite images are shown below:

Figure 3. **Sprite for AI opponent**

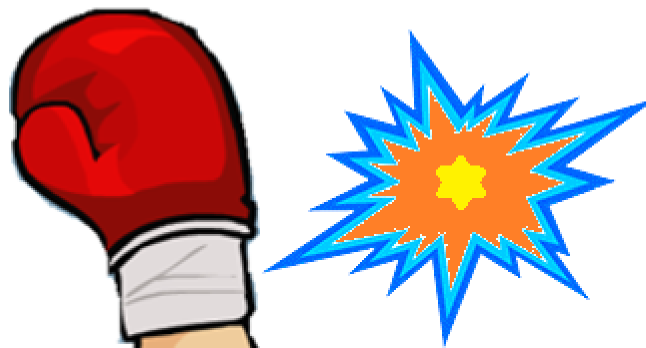


Figure 4. **Sprites for attacking fist and hit effect**

As we can see from these examples, the shapes of most sprites are not restricted in a rectangle. However, when converting RGB image into a mif file, the whole image including the margin is encoded into the file. In order to get rid of the margin part, I add a signal NBLANK to each of these sprites to show the pixel value read from the corresponding ROM is not white, and use this signal in the priority controller to help decide which sprite to display.

The following table contains the sprites I used in the game:

Sprite	Amount	Pixel Size	Total ROM
Fist_atk	1	170*125	63750
Fist_def	1	170*125	63750
Opponent	1	300*200	180000
Num 1	1	96*65	18720
Num 2	1	96*65	18720
Num 3	1	96*65	18720
Start_title	1	25*230	17250

Game_title	1	65*350	68250
Spark	1	120*150	54000
Opponent_fist	1	115*115	39675
Background_slice	1	125*1	375
Total	11	181070	543210

And the priority of the sprites is shown as follows:

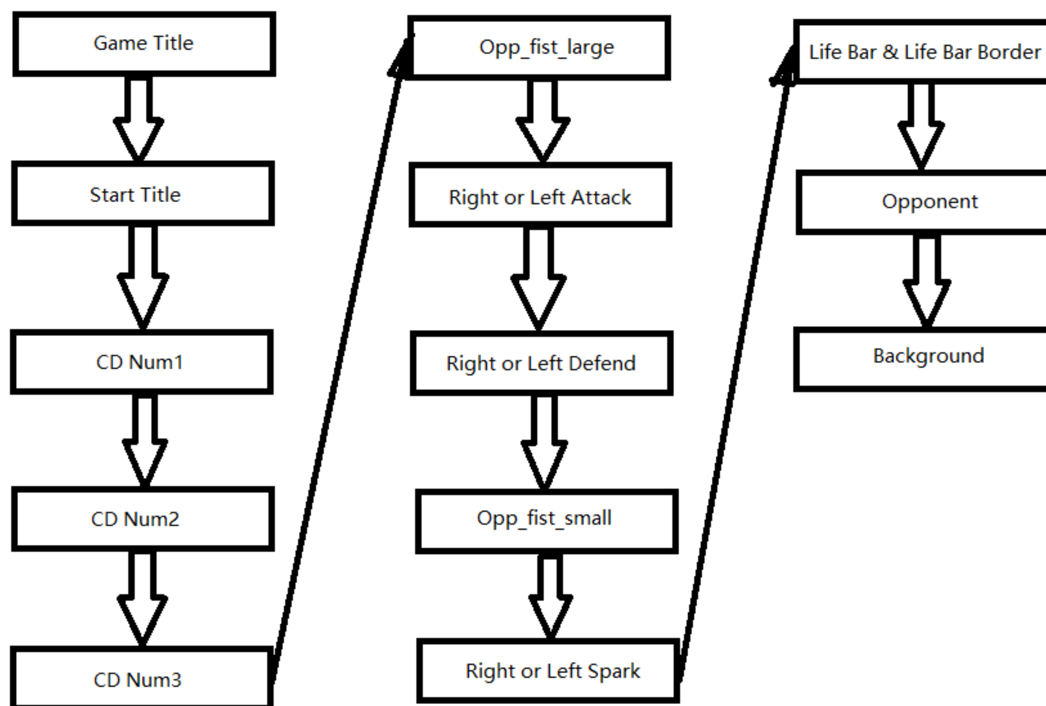


Figure 5. **Sprite Priority**

2.2 Techniques to Save Memory

The pins on the Sockit board is truly limited, during this project, I run into the trouble due to insufficient memory several times while compiling. One reason is that most sprites I used are relatively larger than sprites used in other project. Also I will need to add audio data into the game. Therefore, some methods to help me save memory are necessary.

The first trick I played while implementing sprite graphics is using the symmetricity of the image. There are many images in this game that need to be displayed in a symmetric counterpart, like the attacking fist, the defending fist and the spark. If we just use another ROM for a symmetric counterpart of the image, the memory required is doubled. So the idea here is using one ROM to show both left and right fists or sparks. The way to implement this is: when we compute the address for the ROM, we use both its coordinate in the screen and in its single image. Thus, we only need to use the number of columns to subtract the number of the current pixel

column index instead of using the number of the current pixel column index directly. Figure 6 shows some symmetricity examples.

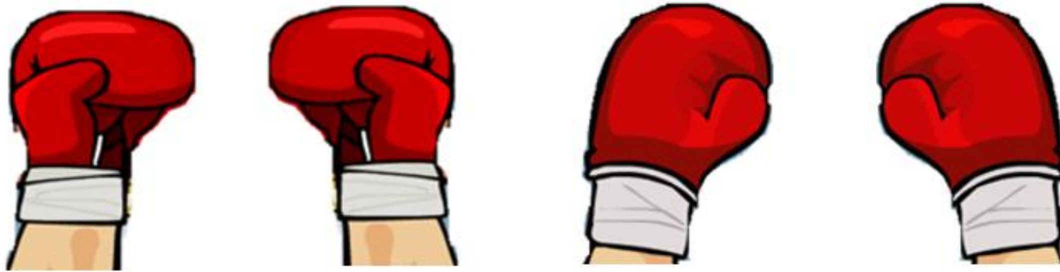


Figure 6. **Symmetricity Examples**

The second technique is using image interpolation to dilate a small image. This is used in the case of opponent's attacking fist (see in Figure 7.). I just stored the 115*115 pixel image in the ROM, and use it for both 115*115 and 230*230 sprite display. And this is implemented by something like down sampling. When we compute the coordinate for the pixel, I use the whole vcount and hcount[10:1] to denote the position in the screen. Here, I omit the last bit of both hcount and vcount, in this way, two pixels at both x and y directions are taken as one. And the resulting image is dilated by a factor of 4. One drawback of this method is that the image could only be dilated by a factor that is the power of 4 (in a single direction, the dilation factor is 2).

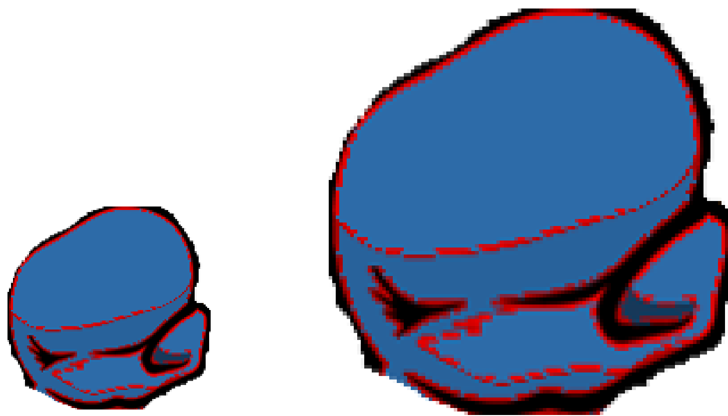


Figure 7. **Samples Using Dilation**

The third way is reduce color information redundancy. As mentioned in the last section, some sprites does not require a ROM if they meet those two conditions. However, for some sprites that only need the shape information, we could also do something to help save memory. Like in the case of sprites of the counting-down number, I merely want to display them in a single color. Therefore, instead of 24 bits ROM (8 bits for each RGB channel) data width, I just use one bit for each pixel to record the shape information in the ROM. This is also the case for the game_title and the starting_title.

Figure 8. **Images Require Only Shape Information**

3. SSM2603 Audio Codec Module

In this project, I used analog devices chip SSM2603 audio CODEC provided by the Cyclone V Sockit board for Audio encoder and decoder. SSM2603 supports sampling rates ranging from 8 kHz to 96 kHz. According to Nyquist sampling theory and human's acceptable audio frequency range, I choose the sampling rate to be 44.1 kHz.

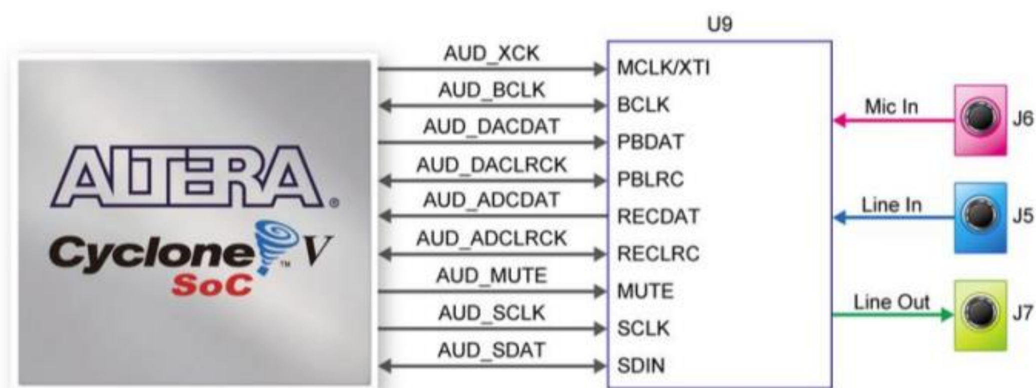


Figure 9. **SSM2603 Audio Codec**

Before the Audio Codec can start capturing or playing audio, it has to be configured with options such as the sampling rate and sample width (how many bits each sample is). The protocol the audio codec uses for configuration is the Inter-Integrated Circuit (I^2C) protocol. This is a two-wire protocol originally designed by Phillips Semiconductor in the 1980s to connect peripherals to the CPU in TV sets. Nowadays it's used to connect low-speed peripherals in all sorts of devices. You can find detailed information about the I^2C protocol at Embedded Systems Academy, from which I took all the timing diagrams you will see below.

Now we have a way to send data through the configuration interface, but what data do we send? For that we must consult the datasheet. The first thing we need to know is what the slave address of the audio codec is. Page 17 of the datasheet states if

the CSB pin is set to 0, the address selected is 0011010; if 1, the address is 0011011. You can figure out what CSB is set to on the SoCKit by looking at the schematic. It's actually a bit ambiguous in the schematic, which shows that CSB is connected to both V_{dd} and ground through resistors. However, the schematic helpfully notes that the default I^2C address is 0x34 / 0x35. Note that this is an 8-bit word. The difference in the LSB refers to the read/write bit. Since we are only writing, the address we use is 0x34. This means CSB must be set to 0.

Now we need to figure out what the 16-bit data words are. 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. In particular, register 8 controls the sampling rate. We want a 44.1 kHz rate for both the ADC and the DAC. According to table 30 in the datasheet, this means we should set CLKODIV2 and CLKDIV2 to 0, SR to 1000, the base oversampling rate to 0, and USB mode select to 0.

As for audio clock, the first clock we have to worry about generating is the master clock MCLK. According to table 30, the frequency of this clock should be 11.2896 MHz. This frequency cannot be generated by simply dividing the master clock (there is no integer number N such that 50 / N is close enough to 11.2896). Fortunately, the Cyclone V contains specialty circuits called Phase-Locked Loops (PLLs) which can generate very precise clock signals. You can add a PLL to your design using Megawizard. The PLL megafunction is under "PLL" -> "Altera PLL v13.1". In the main page of the wizard, change the reference clock frequency to 50 MHz and uncheck the "Enable locked output port" option. In the "Output Clocks" section, change "Number of Clocks" to 2. We will use the PLL to generate a 11.2896 MHz clock for the audio codec and a 50 MHz main clock. Enter these frequencies in for "Desired Frequency". For the 50 MHz clock, you will also need to change the actual frequency to the one right below 50 MHz (you can't generate a clock faster than 50 MHz from a 50 MHz reference). At the beginning of the game, the software extracts the background music sample previously stored in the SDRAM and loads it into the circular queue in the Audio Controller Module, which will continuously send signals to the SSM2603 Audio CODEC to play the sounds until the player exit the game. During the boxing game, when logic control module decides the player's and AI boxer's states, it sends audio controlling signals as well as image controlling signals. It will instruct the DRAM to load the corresponding special sound effect into FIFO, and instruct SSM2603 to play the special sound.

4. Hardware Interface

This part of the project deals with how controlling data are transformed and decoded in software and hardware. In general, there are 14 different controlling signals for different sprites or audios. Each time the logic control unit sends a 32-bits controlling data to the hardware, the 32-bits writedata is divided into 14 parts and

FPGA would change displaying images and audios according to the corresponding controlling signals. Below is a table denoting the detail of writedata components.

Bits in writedata	Annotation	Description
0-2	Opp	Position of opponent
3-4	Spark	States of spark
5-6	My_state	States of player
7	isLeft	Left or right attack
8-10	Attack_state	Position of attacking fist
11-12	My_iddle	Position of idle fist
13	Def_state	Position of defending fist
14-15	Opp_fist	State of opponent's fist
16-17	Audio_state	Which audio is chosen
18-21	My_health_state	My life bar
22-25	Opp_health_state	Opponent's life bar
26	Game_title	State of game title
27	Start_title	State of start title
28-29	Num_state	Counting down num

Besides, I also list the detailed decoding table for each of these control signals.

Opp_state		
Binary	Decimal	Decoding
000	0	Original position
001	1	-5
010	2	5
011	3	-40
100	4	40
101	5	-80
110	6	80
111	7	Dead

Spark_state		
Binary	Decimal	Decoding
0x	0 or 1	None
10	2	Left
11	3	Right

My_state		
Binary	Decimal	Decoding
00	0	Attack
01	1	Defend
10	2	Idle
11	3	Hurt

ifLeft		
Binary	Decimal	Decoding
0	0	Right
1	1	Left

atk_state		
Binary	Decimal	Decoding
000	0	Coordinate 0
001	1	Coordinate 1
010	2	Coordinate 2
011	3	Coordinate 3
100	4	Coordinate 4
101	5	Coordinate 5
110	6	Coordinate 6
111	7	Coordinate 7

Defend_state		
Binary	Decimal	Decoding
0	0	Up
1	1	Down

Opp_fist_state		
Binary	Decimal	Decoding
0x	1 or 0	None
10	2	Small
11	3	Large

Audio_state		
Binary	Decimal	Decoding
00	0	Hurt
01	1	Dodge
10	2	Defend
11	3	None

Game & Start_title		
Binary	Decimal	Decoding
0	0	Hidden
1	1	Shown

Num_state		
Binary	Decimal	Decoding
00	0	None
01	1	Cd num1

10	2	Cd num2
11	3	Cd num3

As for my & opp_health_state, the 4-bits signal denotes exactly the number of health remained for the player/opponent, thus it is not listed in detail.

5. Logic Control Unit

For Logic Control Module, I intend to run the main part of the program in a while loop. In each iteration, control module first checks the keyboard input and decides the current player's state through the keyboard input. Using the function RAN(), Logic Control Module would decide what AI boxer will act according to player's current state. Both player and AI boxer would have four states: attack, idle, defend and hurt. Once the states of player and AI are decided, Logic Control Module would generate a 32 bits controlling data and send this data to hardware interface through Device Driver Module.

AI boxer would do idle, attack or defend according to player's idle or defend states, defend or hurt according to player's attack state. If AI boxer chooses to attack, control unit would set player's state to hurt. And when one side's state becomes defend or hurt and the other side's state is attack, the display condition is met and control unit would send data to device driver, after that both player's and AI boxer's states are reset to idle.

This is shown in the following logic control flowchart.

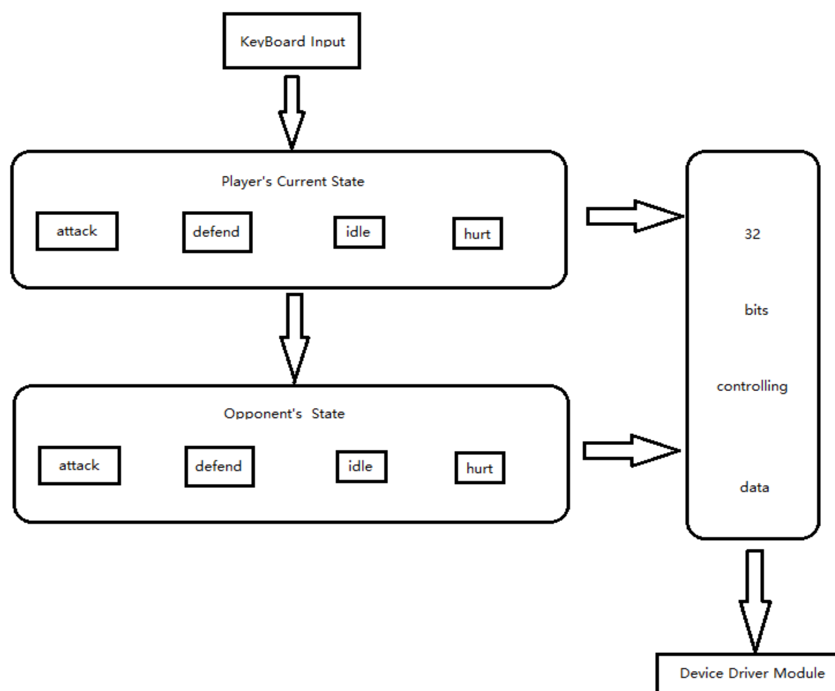


Figure 10. **Logic Control Unit**

6. Device Driver Unit

In this project, users could interact with the AI opponent with a USB-keyboard. The driver of the USB-keyboard could be implemented through libusb if you could successfully set up the Linux environment. But in this project, I would like to introduce another method to read in and decode USB-keyboard input.

Once a USB-keyboard is connected to the Linux, the system automatically creates an input file denoted as 'event0' at /dev/input. We could use the 'cat' command to check the file's content while we press some buttons on the USB-keyboard. This simple experiment shows that we could get the USB-keyboard input by directly reading this 'event0' file.

In linux, we just need to include 'linux/input.h' for our head files, and then we could call the function 'open("/dev/input/event0", O_RDONLY)' and 'read(usbkeyboard_fd, &usbkeyboard_buff, sizeof(struct input_event))' to store the USB-keyboard input into a variable 'usbkeyboard_buff'.

And there are some problems concerning decoding the stored input data. In the function, every time a button is pressed, the function generates six input data. So we need a counter to distinguish one input and six inputs. More detailed code is listed in the appendix.

7. Game Demo

In this section I list several screen shots while the game program is running.

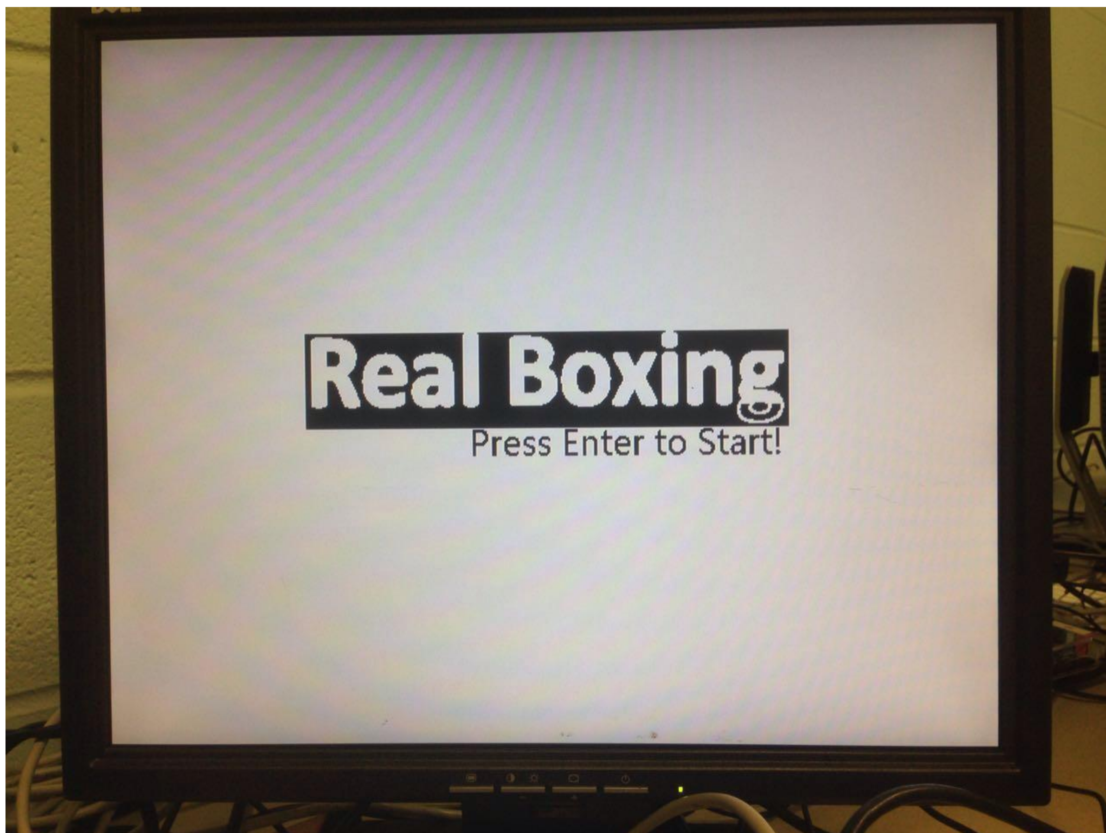


Figure 11. **Game Start Screen**

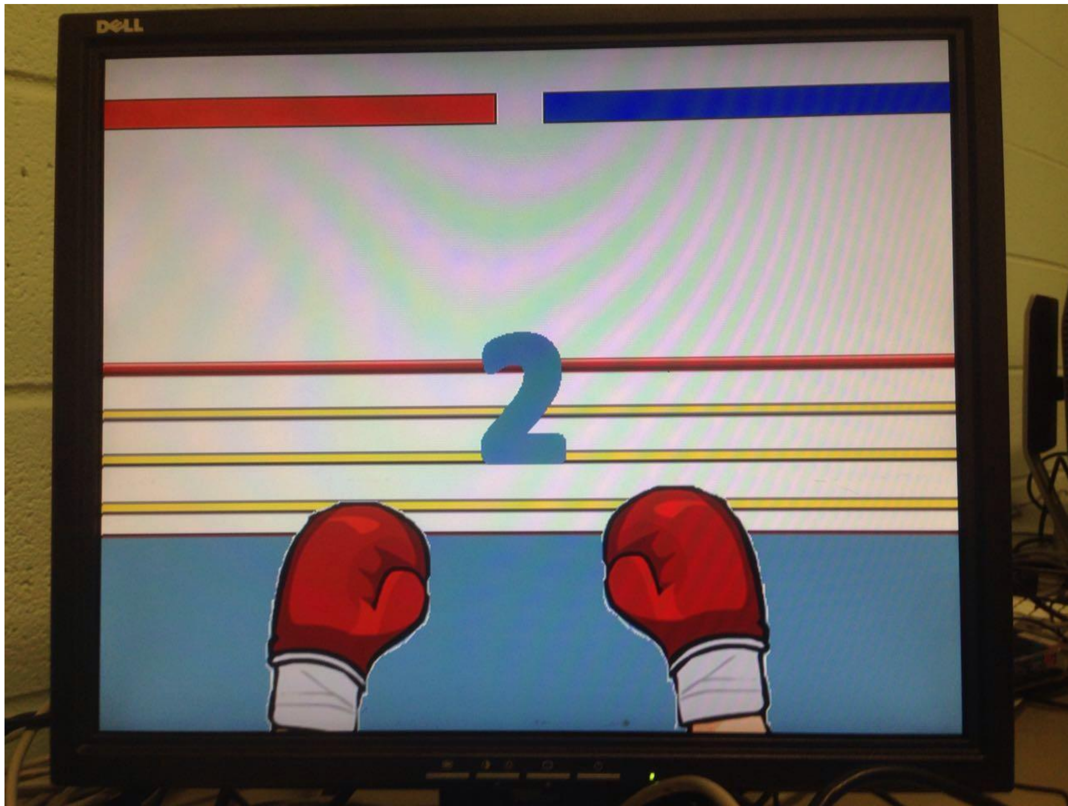


Figure 12. **Counting Down to Start Screen**

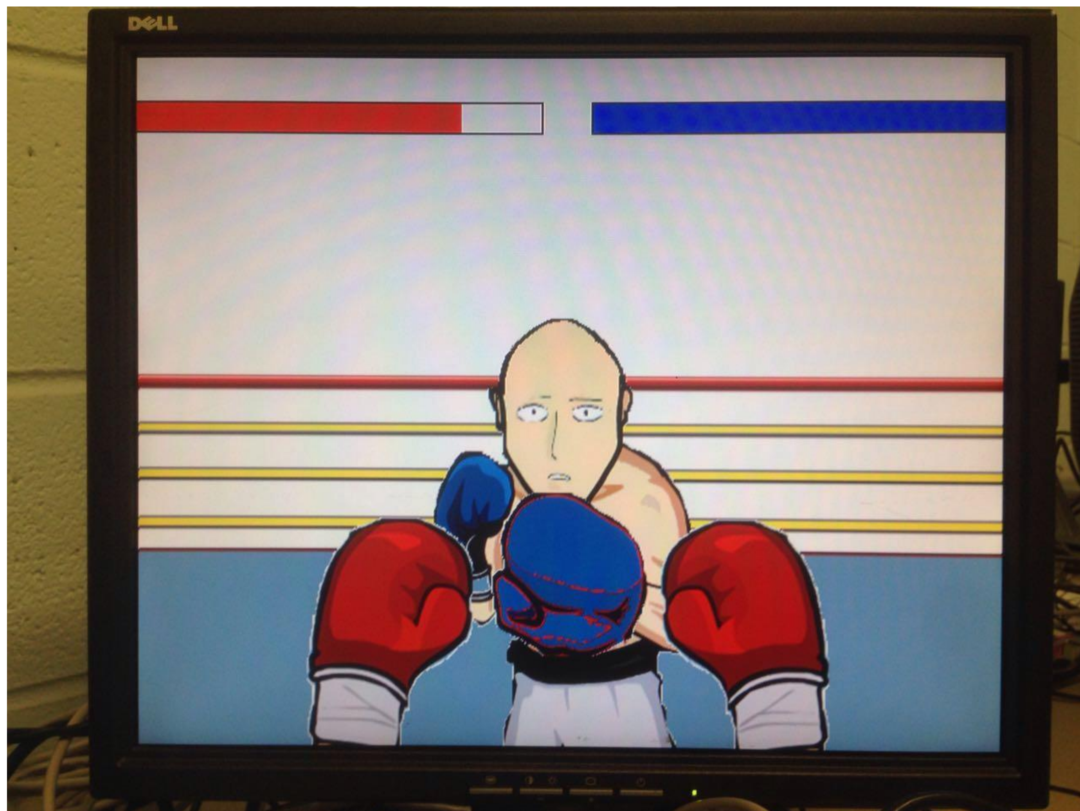


Figure 13. **Opponent Attack Screen**

8. Summary and Future Direction

8.1 Summary

The most difficult part for me is still the game logic design. Unlike hardware design like sprite graphics or SSM2603 audio codec, I could hardly find any reference online or from previous projects for the software design. Besides, I got a lot of trouble while dealing with USB-keyboard driver. The function 'read()' from 'linux/input.h' returns to me a quite strange format of input data. I spent much time trying to decoding the keyboard input.

Still, I am kind of proud that I was able to finish the task on time by myself. I always want to do something stupid but impressive in Columbia. That would be worth memorizing after I graduate.

8.2 Future Direction

(1) One thing I really want to implement in this project is using an acceleration sensor as controller. We could attach such sensors on our fist. When we punch or defend, the sensor captures the speed and direction of our fist and uses these data to determine attacking damage.

(2) Another intriguing direction of this project is to design a more intelligent computer opponent. Currently the state machine contains only three states for the AI boxer. We could design more complicated model and give it more reflection than just dodge and attack.

9. Appendix

9.1 Software

9.1.1 boxing.c

```
#include "usbkeyboard.h"
#include "vga_led.h"
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <stdio.h>
#include <sys/ioctl.h>
```

```c
#define ATK_INTV 25000 //25000
#define BLINK_INTV 200000
#define IDL_INTV 100000 //100000
#define DFD_INTV 200000
//set AI boxer parameter here
#define DDG_PROB 7  //7
#define ATK_PROB 3  //3
#define REF_INTV 300000




/*************************************************************game
variables*********************************************/

//keyboard variables
int key_value = 0;  //keyboard read in value
int insign=0;       //keyboard pressed or idle
int temp_insign=0;

//vga led variables
int vga_led_fd;
unsigned int other_state;


//game state variables
int myhealth_state=15;
int opphealth_state = 15;
int game_on=0;


/**************************************************************system
functions*********************************************/

//KeyBoard Thread
pthread_t keyboard_thread;
void *keyboard_thread_f(void *ignore)
      {
     while(1)
       {
       //every time a key is pressed, this part is called six times, which means 6 same
```

```
key_values are given back.
        // to distinguish the same key_value comes from the same key or no key press at all,
use insign to count.
        insign += 1;
        usbkeyboard(&key_value);
      //  printf("insign: %d\n",insign);
        }
      }
```

```
//vga led initialize function
```

```
void init_vga_led()
{
  static const char filename[] = "/dev/vga_led";

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

```
unsigned int gen_other_state(unsigned int audio_state,
                                  unsigned int myhealth_state,
                                  unsigned int opphealth_state,
                                  unsigned int title_state,
                                  unsigned int start_state,
                                  unsigned int num_state)
{
  unsigned int other_state;

  other_state = audio_state | myhealth_state << 2 | opphealth_state << 6 | title_state <<
10 | start_state << 11 | num_state << 12;
  return (other_state);
}
```

```
//vga ioctl write function

void write_game_state(unsigned int opp_state,
```

```
                      unsigned int spark,
               unsigned int my_state,
               unsigned int isleft,
               unsigned int atk_state,
               unsigned int idle_state,
               unsigned int def_state,
               unsigned int opp_fist,
                      unsigned int other_state
                      )
{
  game_state gs;
  gs.opp_state = opp_state;
  gs.spark = spark;
  gs.my_state = my_state;
  gs.isleft = isleft;
  gs.atk_state = atk_state;
  gs.idle_state = idle_state;
  gs.def_state = def_state;
  gs.opp_fist = opp_fist;
  gs.other_state = other_state;

  if(ioctl(vga_led_fd, VGA_LED_WRITE_GAME_STATE, &gs)){
    // perror("ioctl(VGA_LED_WRITE_GAME_STATE) failed");
  }
}




/*************************************************************display
functions*******************************************************/

void left_atk_hit()
{
  int cnt1,cnt2;
  //                    opp  spark  mys  isleft  atk  idle  def  oppfist
audio  myhealth  opphealth  title  start  cdnum
  for(cnt1=0; cnt1<7; cnt1++)
    {
    write_game_state(  2,     0,     0,      1, cnt1,     0,    0,         0,
gen_other_state(3, myhealth_state, opphealth_state, 0,    0,    0) );
    usleep(ATK_INTV);
    }
  opphealth_state -= 1;
```

```
  write_game_state(  2,     2,        0,       1,      7,      0,     0,       0,
gen_other_state(0, myhealth_state, opphealth_state,0,0,0));
  usleep(5*ATK_INTV);

  for(cnt2=6; cnt2>=0; cnt2--)
    {
    write_game_state(  0,     0,        0,       1,  cnt2,   0,     0,       0, gen_other_state(3,
myhealth_state, opphealth_state,0,0,0));
    usleep(ATK_INTV);
    }

}

void right_atk_hit()
{
  int cnt1,cnt2;
  //                 opp spark mystate isleft    atkstate   idlestate  defstate oppfist
audio   myhealth  opphealth  title  start  cdnum
  for(cnt1=0; cnt1<7; cnt1++)
    {
    write_game_state(  1,     0,        0,       0,         cnt1,            0,          0,       0,
gen_other_state(3,myhealth_state, opphealth_state,0,0,0));
    usleep(ATK_INTV);
    }
  opphealth_state -= 1;
  write_game_state(  1,     3,        0,       0,           7,             0,          0,          0,
gen_other_state(0, myhealth_state, opphealth_state,0,0,0));
  usleep(5*ATK_INTV);

  for(cnt2=6; cnt2>=0; cnt2--)
    {
    write_game_state(  0,     0,        0,       0,         cnt2,            0,          0,       0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0));
    usleep(ATK_INTV);
    }

}

void left_atk_ddg()
{
  int cnt1,cnt2;
  //                 opp   spark   mys   isleft  atk   idle   def   oppfist
audio  myhealth   opphealth title  start  cdnum
  for(cnt1=0; cnt1<3; cnt1++)
```

```
    {
        write_game_state(  2,     0,     0,      1,  cnt1,     0,    0,         0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0) );
        usleep(ATK_INTV);
    }

    write_game_state(  4,     0,        0,      1,     3,     0,    0,       0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0));
    usleep(2*ATK_INTV);

    for(cnt1=4; cnt1<7; cnt1++)
    {
        write_game_state(  4,     0,     0,      1,  cnt1,     0,    0,         0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0) );
        usleep(ATK_INTV);
    }

    write_game_state(  6,     0,        0,      1,     7,     0,    0,       0,
gen_other_state(1, myhealth_state, opphealth_state,0,0,0));
    usleep(4*ATK_INTV);

    for(cnt2=6; cnt2>3; cnt2--)
    {
        write_game_state(  4,     0,        0,      1,  cnt2,   0,    0,       0, gen_other_state(3,
myhealth_state, opphealth_state,0,0,0));
        usleep(ATK_INTV);
    }

    write_game_state(  4,     0,        0,      1,     3,     0,    0,       0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0));
    usleep(2*ATK_INTV);

    for(cnt2=3; cnt2>=0; cnt2--)
    {
        write_game_state(  2,     0,        0,      1,  cnt2,   0,    0,       0, gen_other_state(3,
myhealth_state, opphealth_state,0,0,0));
        usleep(ATK_INTV);
    }

}

void right_atk_ddg()
{
    int cnt1,cnt2;
```

```
  //                 opp  spark  mys  isleft  atk  idle  def  oppfist
audio  myhealth  opphealth
  for(cnt1=0; cnt1<3; cnt1++)
    {
    write_game_state(  1,     0,     0,      0, cnt1,     0,    0,        0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0) );
    usleep(ATK_INTV);
    }

  write_game_state(  3,     0,       0,      0,     3,     0,    0,       0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0));
  usleep(2*ATK_INTV);

  for(cnt1=4; cnt1<7; cnt1++)
    {
    write_game_state(  3,     0,     0,      0, cnt1,     0,    0,        0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0) );
    usleep(ATK_INTV);
    }

  write_game_state(  5,     0,       0,      0,     7,     0,    0,       0,
gen_other_state(1, myhealth_state, opphealth_state,0,0,0));
  usleep(4*ATK_INTV);

  for(cnt2=6; cnt2>3; cnt2--)
    {
    write_game_state(  3,     0,       0,      0, cnt2,   0,    0,       0, gen_other_state(3,
myhealth_state, opphealth_state,0,0,0));
    usleep(ATK_INTV);
    }

  write_game_state(  3,     0,       0,      0,     3,     0,    0,       0,
gen_other_state(3, myhealth_state, opphealth_state,0,0,0));
  usleep(2*ATK_INTV);

  for(cnt2=3; cnt2>=0; cnt2--)
    {
    write_game_state(  1,     0,       0,      0, cnt2,   0,    0,       0, gen_other_state(3,
myhealth_state, opphealth_state,0,0,0));
    usleep(ATK_INTV);
    }

}
```

```
void dfd_opphit()
{
	write_game_state(0,0,1,0,0,0,1,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
	usleep(DFD_INTV);
	write_game_state(0,0,1,0,0,0,1,2,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
	usleep(DFD_INTV);
	write_game_state(0,0,1,0,0,0,0,3,gen_other_state(2,myhealth_state,
opphealth_state,0,0,0));
	usleep(DFD_INTV*0.5);
	write_game_state(0,0,1,0,0,0,0,3,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
	usleep(DFD_INTV*0.5);
}


void dfd_oppidle()
{
	write_game_state(1,0,1,0,0,0,1,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
	usleep(DFD_INTV);
	write_game_state(0,0,1,0,0,0,1,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
	usleep(DFD_INTV);
	write_game_state(2,0,1,0,0,0,1,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
	usleep(DFD_INTV);
	write_game_state(0,0,1,0,0,0,1,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
}


void my_idle1()
{
    write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
    usleep(IDL_INTV);
    write_game_state(0,0,2,0,0,1,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
    usleep(IDL_INTV);
    write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
```

```
opphealth_state,0,0,0));
}

void my_idle2()
{
    write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
    usleep(IDL_INTV);
    write_game_state(0,0,2,0,0,2,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
    usleep(IDL_INTV);
    write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
}


void player_down()
{
        write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(0,0,3,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(0,0,3,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(0,0,3,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
}


void AI_down()
{
        write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
```

```
        write_game_state(7,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(7,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(0,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
        write_game_state(7,0,2,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
        usleep(BLINK_INTV);
}

void countdown()
{
      write_game_state(7,0,0,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,3));
      usleep(1000000);
      write_game_state(7,0,0,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,2));
      usleep(1000000);
      write_game_state(7,0,0,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,0,0,1));
      usleep(1000000);
}

/**************************************************main
function*************************************************************/

int main()
{
  //initialize devices
  init_keyboard();
  init_vga_led();
  //initialize game state
  write_game_state(7,0,3,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,1,1,0));


  //keyboard thread start
```

```
  pthread_create(&keyboard_thread, NULL, keyboard_thread_f, NULL);


/***********************************************main
loop*************************************************/
  while(1)
      {
        write_game_state(7,0,3,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,1,1,0));
        usleep(350000);
        if(insign>temp_insign && key_value == 28)
          {
            // insign = 0;
            // temp_insign = insign;
             myhealth_state = 15;
             opphealth_state = 15;
             game_on = 1;

             countdown();


        while(game_on == 1)
          {
        if(insign>temp_insign)//check whether a key is pressed
           {
           printf("\nkey_value: %d\n\n",key_value);
         // temp_insign+=6;
           if(key_value == 105)
              {
              int jud = 1+(int)(10.0*rand()/(RAND_MAX+1.0));
              printf("jud: %d\n",jud);
                if(jud>DDG_PROB)
                 left_atk_hit();
                else
                 left_atk_ddg();
              }
           else if(key_value == 106)
              {
              int jud = 1+(int)(10.0*rand()/(RAND_MAX+1.0));
              printf("jud: %d\n",jud);
                if(jud>DDG_PROB)
                 right_atk_hit();
                else
                 right_atk_ddg();
```

```
            }
            else if(key_value == 108 || key_value == 57)
            {
                int jud = 1+(int)(10.0*rand()/(RAND_MAX+1.0));
                if(jud > 8)
                {
                  dfd_opphit();
                }
                else
                {
                  dfd_oppidle();
                }
            }
            else if(key_value==1)
                {
                game_on = 0;
                    key_value = 2;
                }
            temp_insign = insign;
        }

        else //no key is pressed
        {
            int jud = 1+(int)(10.0*rand()/(RAND_MAX+1.0));
            if(jud>ATK_PROB)
            {
                my_idle1();
                if(insign>temp_insign)
                continue;
                my_idle2();
                if(insign>temp_insign)
                continue;
            }
            else
            {
                    usleep(0.5*REF_INTV);
                write_game_state(0,0,2,0,0,0,0,2,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
                usleep(REF_INTV);
                if(insign>temp_insign && (key_value == 108 || key_value == 57))
                {
                   write_game_state(0,0,1,0,0,0,1,3,gen_other_state(2,myhealth_state,
opphealth_state,0,0,0));
                   usleep(DFD_INTV*0.5);
```

```
                write_game_state(0,0,1,0,0,0,0,3,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
                usleep(DFD_INTV*0.5);
                temp_insign = insign;
                }
             else
                {
                myhealth_state -= 1;
                write_game_state(0,0,3,0,0,0,0,3,gen_other_state(0,myhealth_state,
opphealth_state,0,0,0));
                usleep(DFD_INTV*0.5);
                write_game_state(0,0,3,0,0,0,0,3,gen_other_state(3,myhealth_state,
opphealth_state,0,0,0));
                usleep(DFD_INTV*0.5);
                temp_insign = insign;
                }
             }
          }

      //temp_insign must be updated before the end of one loop
       if(myhealth_state == 0 || opphealth_state ==0)
          {
          if(myhealth_state == 0)
            player_down();
          else
            AI_down();
          game_on = 0;
          }
       if(key_value==1)
          game_on = 0;

       }//while(game_on ==1)

     }//if press enter to start
   else if(insign>temp_insign && key_value == 1)
      break;
   write_game_state(7,0,3,0,0,0,0,0,gen_other_state(3,myhealth_state,
opphealth_state,1,0,0));
   usleep(350000);
  }//while(1)


/**********************************program ends
here********************************************/
```

```
  printf("program ended\n");

  pthread_cancel(keyboard_thread);
  pthread_join(keyboard_thread, NULL);

  return 1;
}
```

9.1.2 usbkeyboard.c

```
#include "usbkeyboard.h"

struct input_event usbkeyboard_buff;
int usbkeyboard_fd;
int usbkeyboard_read_nu;

void init_keyboard()
{

    usbkeyboard_fd = open("/dev/input/event0", O_RDONLY);
    if (usbkeyboard_fd < 0)
    {
        perror("can not open device usbkeyboard!");
        exit(1);
    }

}



void usbkeyboard(int *temp)
{
//current problem: cannot check status where no button is pressed, not sure what value read
gives back to denote no button
//feel like is no key is pressed, the read will loop until a key is found.
    if(read(usbkeyboard_fd,&usbkeyboard_buff,sizeof(struct input_event)) != 16)
      { printf("nothing read in\n"); }
    else
      {
      if(usbkeyboard_buff.type==1 & usbkeyboard_buff.value==1)
        {
     //     printf("type:%d code:%d
value:%d\n",usbkeyboard_buff.type,usbkeyboard_buff.code,usbkeyboard_buff.value);
         *temp = usbkeyboard_buff.code;
```

```
        }
   
//printf("temp: %d\n",*temp);
      
      }
}
```

9.1.3 usbkeyboard.h

```
#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <linux/input.h>

//extern void usbkeyboard();

#endif
```

9.1.4 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_led.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree vga_led.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_led"

/*
 * Information about our device
 */
struct vga_led_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_led_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
```

```
{
    vga_led_arg_t vla;
        circle_center cc;
        game_state gs;

    switch (cmd) {
    case VGA_LED_WRITE_DIGIT:
         if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                      sizeof(vga_led_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_led_arg_t *) arg,
                      sizeof(vga_led_arg_t)))
              return -EACCES;
         if (vla.digit > 8)
              return -EINVAL;
         vla.segments = dev.segments[vla.digit];
         if (copy_to_user((vga_led_arg_t *) arg, &vla,
                    sizeof(vga_led_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 | cc.y<<3 | cc.z<<5), dev.virtbase); // generate
writedata[31:0], x=writedata[19:10], y=writedata[9:0]
              break;
        
        case VGA_LED_WRITE_GAME_STATE:
             if (copy_from_user(&gs, (game_state *) arg, sizeof(game_state)))
                return -EACCES;
             unsigned int test = gs.opp_state | gs.spark << 3 | gs.my_state << 5 | gs.isleft
<< 7 | gs.atk_state << 8 |
                         gs.idle_state << 11 | gs.def_state << 13 | gs.opp_fist << 14 |
gs.other_state << 16;
             iowrite32( test , dev.virtbase);

    default:
```

```
        return -EINVAL;
    }

    return 0;
}

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

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

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init vga_led_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_led */
    ret = misc_register(&vga_led_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_led_misc_device);
	return ret;
}

/* Clean-up code: release resources */
static int vga_led_remove(struct platform_device *pdev)
{
	iounmap(dev.virtbase);
	release_mem_region(dev.res.start, resource_size(&dev.res));
	misc_deregister(&vga_led_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_led_of_match[] = {
	{ .compatible = "altr,vga_led" },
	{},
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_led_driver = {
	.driver	= {
		.name	= DRIVER_NAME,
```

```
        .owner    = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_led_of_match),
    },
    .remove   = __exit_p(vga_led_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_led_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

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

module_init(vga_led_init);
module_exit(vga_led_exit);

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

9.1.4 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_led_arg_t;

typedef struct{
  unsigned int x;   //32 bits unsigned int x coordinate of the ball
```

```
  unsigned int y;                       //y coordinate of the ball
  unsigned int z;
} circle_center;

typedef struct{
  unsigned int opp_state;
  unsigned int spark;
  unsigned int my_state;
  unsigned int isleft;
  unsigned int atk_state;
  unsigned int idle_state;
  unsigned int def_state;
  unsigned int opp_fist;
  unsigned int other_state; //other state contains audio states, my&opp states
} game_state;

#define VGA_LED_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_LED_WRITE_DIGIT _IOW(VGA_LED_MAGIC, 1, vga_led_arg_t *)
#define VGA_LED_READ_DIGIT  _IOWR(VGA_LED_MAGIC, 2, vga_led_arg_t *)
#define VGA_LED_WRITE_CENTER _IOW(VGA_LED_MAGIC, 3, vga_led_arg_t *)
#define VGA_LED_WRITE_GAME_STATE _IOW(VGA_LED_MAGIC, 4, vga_led_arg_t *)

#endif
```

9.1.5 Makefile

```
# compile from fpga sw file
hello : boxing.o usbkeyboard.o
    cc -Wall -o boxing boxing.o usbkeyboard.o -pthread

hello.o : boxing.c usbkeyboard.h

usbkeyboard.o : usbkeyboard.c usbkeyboard.h
```

9.2 Hardware

9.2.1 VGA_LED.sv

```
/*
```

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

module VGA_LED(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,

                  output logic [1:0] VGA_audio_control,
                  output logic [1:0] Test

                  );



    logic  [2:0] opp_state;
    logic  [1:0] spark_state;
   logic  [1:0] my_state;
   logic  isleft;
   logic  [2:0] atk_state;
   logic  [1:0] idle_state;
   logic  def_state;
    logic  [1:0] opp_fist_state;

    logic  [3:0] myhealth_state;
    logic  [3:0] opphealth_state;

    logic  title_state;
    logic  start_state;

    logic  [1:0] num_state;

    assign VGA_audio_control = writedata[17:16];
    assign Test = writedata[1:0];

```

```
    VGA_LED_Emulator led_emulator(.clk50(clk), .*);

    always_ff @(posedge clk)
     begin
       if (reset)
            begin
              opp_state <= 3'b111;
              spark_state <= 2'b00;
              my_state <= 2'b11;
              isleft <= 1;
              atk_state <= 3'b111;
              idle_state <= 2'b00;
              def_state <= 0;
              opp_fist_state <= 2'b00;
              myhealth_state <= 4'b1111;
              opphealth_state <= 4'b1111;
              title_state <= 1;
              start_state <= 1;
           num_state <= 2'b00;
           end
          else if (chipselect && write)
           begin
              opp_state <= writedata[2:0];
              spark_state <= writedata[4:3];
              my_state <= writedata[6:5];
              isleft <= writedata[7];
              atk_state <= writedata[10:8];
              idle_state <= writedata[12:11];
              def_state <= writedata[13];
              opp_fist_state <= writedata[15:14];
              myhealth_state <= writedata[21:18];
              opphealth_state <= writedata[25:22];
           title_state <= writedata[26];
              start_state <= writedata[27];
              num_state <= writedata[29:28];
           end
         end

endmodule
```

9.2.2 VGA_LED_Emulator.sv

```
/*
```

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

module VGA_LED_Emulator(
 input logic     clk50, reset,

/*
 input logic  [23:0] atk_left_pos,
 input logic  [23:0] atk_right_pos,
*/
 input logic  [2:0] opp_state,
 input logic  [1:0] spark_state,

 input logic  [1:0] my_state,
 input logic  isleft,
 input logic  [2:0] atk_state,
 input logic  [1:0] idle_state,
 input logic  def_state,
 input logic  [1:0] opp_fist_state,

 input logic  [3:0] myhealth_state,
 input logic  [3:0] opphealth_state,

 input logic  title_state,
 input logic  start_state,

 input logic  [1:0] num_state,

 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




    ////////////////START HERE/////////////////////


    /*******************************data read in*******************************/

    //my health state --- 4bits
    logic [3:0] myhealth_state_temp;

    //opponent health state --- 4bits
    logic [3:0] opphealth_state_temp;

    //opponent position---3bits
    logic [2:0] opp_state_temp;

    //spark---2bits
    logic [1:0] spark_state_temp;

    //my states---2bits
    logic [1:0] my_state_temp;

    //left or right attack---1bits
    logic isleft_temp;
```

```
    //attack position states---3bits
    logic [2:0] atk_state_temp;

    //idle position states---2bits
    logic [1:0] idle_state_temp;

    //defend states---1bits
    logic def_state_temp;

    //opponent fist states---2bits
    logic [1:0] opp_fist_state_temp;

    //game title and start label
    logic title_state_temp;
    logic start_state_temp;

    //count down number state
    logic [1:0] num_state_temp;



        always_ff @(posedge clk50)
        begin
            if(vcount > 10'd480)
                begin
            opp_state_temp <= opp_state;
                    spark_state_temp <= spark_state;
                    my_state_temp <= my_state;
                    isleft_temp <= isleft;
                    atk_state_temp <= atk_state;
                    idle_state_temp <= idle_state;
                    def_state_temp <= def_state;
                    opp_fist_state_temp <= opp_fist_state;
                    myhealth_state_temp <= myhealth_state;
                    opphealth_state_temp <= opphealth_state;
                    start_state_temp <= start_state;
                    title_state_temp <= title_state;
                    num_state_temp <= num_state;
                end
            else
                begin
                    opp_state_temp <= opp_state_temp;
                    spark_state_temp <= spark_state_temp;
```

```
                    my_state_temp <= my_state_temp;
                    isleft_temp <= isleft_temp;
                    atk_state_temp <= atk_state_temp;
                    idle_state_temp <= idle_state_temp;
                    def_state_temp <= def_state_temp;
                    opp_fist_state_temp <= opp_fist_state_temp;
                    myhealth_state_temp <= myhealth_state_temp;
                    opphealth_state_temp <= opphealth_state_temp;
                    start_state_temp <= start_state_temp;
                    title_state_temp <= title_state_temp;
                    num_state_temp <= num_state_temp;
                  end
         end



    /*************************************************************************************
************/


    /*******************************ROM initialization******************************/

    //background
    logic [23:0] data_background;
    logic [6:0] addr_background;
    bg_slice bg_slice(.address(addr_background), .clock(clk50), .q(data_background));


    //attack&idle fist
    logic [23:0] data_atk;
    logic [14:0] addr_atk;
    fist_atk fist_atk(.address(addr_atk), .clock(clk50), .q(data_atk));


    //defend fist
    logic [23:0] data_def;
    logic [14:0] addr_def;
    fist_def fist_def(.address(addr_def), .clock(clk50), .q(data_def));


    //opponent
    logic [23:0] data_opp;
    logic [15:0] addr_opp;
    opponent opponent(.address(addr_opp), .clock(clk50), .q(data_opp));
```

```
    //spark
    logic [23:0] data_spark;
    logic [14:0] addr_spark;
    spark spark(.address(addr_spark), .clock(clk50), .q(data_spark));


   //opponent fist
    logic [23:0] data_opp_fist;
    logic [13:0] addr_opp_fist;
    opp_fist opp_fist(.address(addr_opp_fist), .clock(clk50), .q(data_opp_fist));


    //game title
    logic data_title;
    logic [14:0] addr_title;
    title title(.address(addr_title), .clock(clk50), .q(data_title));

    //start label
    logic data_start;
    logic [12:0] addr_start;
    start start(.address(addr_start), .clock(clk50), .q(data_start));

    //count down number 1-3
    logic data_num1;
    logic [12:0] addr_num1;
    num1 num1(.address(addr_num1), .clock(clk50), .q(data_num1));

   logic data_num2;
    logic [12:0] addr_num2;
    num2 num2(.address(addr_num2), .clock(clk50), .q(data_num2));

    logic data_num3;
    logic [12:0] addr_num3;
    num3 num3(.address(addr_num3), .clock(clk50), .q(data_num3));


    /*****************************************************************************
*******/



    /***********************************sprite
```

```
graphics*********************************/

    //count down number 1-3
    logic num1_on;
    logic num2_on;
    logic num3_on;

    logic num1_nblank;
    assign num1_nblank = !(data_num1==0);


    logic num2_nblank;
    assign num2_nblank = !(data_num2==0);


    logic num3_nblank;
    assign num3_nblank = !(data_num3==0);

    always_comb
      begin
        if(num_state_temp == 2'b01 & hcount[10:1]>=288 & hcount[10:1]<353 & vcount>=200
&vcount<296)
             begin
              num1_on <= 1;
              num2_on <= 0;
              num3_on <= 0;
              addr_num1 <= (hcount[10:1]-288) + (vcount-200)*65;
              addr_num2 <= 0;
              addr_num3 <= 0;
             end
          else if(num_state_temp == 2'b10 & hcount[10:1]>=288 & hcount[10:1]<353 &
vcount>=200 &vcount<296)
             begin
              num1_on <= 0;
              num2_on <= 1;
              num3_on <= 0;
              addr_num2 <= (hcount[10:1]-288) + (vcount-200)*65;
              addr_num1 <= 0;
              addr_num3 <= 0;
             end
        else if(num_state_temp == 2'b11 & hcount[10:1]>=288 & hcount[10:1]<353 &
vcount>=200 &vcount<296)
             begin
              num1_on <= 0;
```

```
                num2_on <= 0;
                num3_on <= 1;
                addr_num3 <= (hcount[10:1]-288) + (vcount-200)*65;
                addr_num2 <= 0;
                addr_num1 <= 0;
                end
            else
                begin
                num1_on <= 0;
                num2_on <= 0;
                num3_on <= 0;
                addr_num1 <= 0;
                addr_num2 <= 0;
                addr_num3 <= 0;
                end
        end



    //start label and game title
    logic start_on;
    logic title_on;

    always_comb
        begin
            if(title_state_temp & hcount[10:1]>=145 & hcount[10:1]<495 & vcount>=200
&vcount<265)
            begin
            title_on <= 1;
            addr_title <= (hcount[10:1]-145) + (vcount-200)*350;
            end
            else
            begin
            title_on <= 0;
            addr_title <= 0;
            end
        end

    always_comb
        begin
            if(start_state_temp & hcount[10:1]>=265 & hcount[10:1]<495 & vcount>=265
&vcount<290)
            begin
            start_on <= 1;
```

```
            addr_start <= (hcount[10:1]-265) + (vcount-265)*230;
            end
            else
          begin
            start_on <= 0;
            addr_start <= 0;
            end
        end



    //health bar border
    logic border_on;

    always_comb
        begin
          if((hcount[10:1]>=0 & hcount[10:1]<=301 & vcount==30) | (hcount[10:1]>=0 &
hcount[10:1]<=301 & vcount==51) |
                    (hcount[10:1]>=338 & hcount[10:1]<=639 & vcount==30) | (hcount[10:1]>=338
& hcount[10:1]<=639 & vcount==51) |
                        (hcount[10:1]==301 & vcount>30 & vcount<51) | (hcount[10:1]==338 &
vcount>30 & vcount<51))
              border_on <= 1;
            else
                border_on <= 0;
        end


    //health bar
    logic myhealth_on;
    logic opphealth_on;


    always_comb
        begin
         if(hcount[10:1]>=0 & hcount[10:1]<=(20*myhealth_state_temp) & vcount>30 &
vcount<51)
             myhealth_on <= 1;
           else
             myhealth_on <= 0;
        end

    always_comb
        begin
```

```
        if(hcount[10:1]<=639 & hcount[10:1]>=(639-20*opphealth_state_temp) & vcount>30 &
vcount<51)
          opphealth_on <= 1;
        else
            opphealth_on <= 0;
      end




    //background
    logic bg_up_on;
    logic bg_mid_on;
    logic bg_btm_on;

    always_comb
      begin
        if(vcount<220)
             begin
             bg_up_on <= 1;
             bg_mid_on <= 0;
             bg_btm_on <= 0;
             addr_background <= 0;
             end
           else if(vcount>=220 & vcount<345)
             begin
             bg_up_on <= 0;
             bg_mid_on <= 1;
             bg_btm_on <= 0;
             addr_background <= (vcount-220);
             end
           else
             begin
             bg_up_on <= 0;
             bg_mid_on <= 0;
             bg_btm_on <= 1;
             addr_background <= 0;
             end
      end



     //opponent
```

```
	logic opp_nblank;
	assign opp_nblank = !(data_opp == 24'd16777215);

	logic opp_on;

	always_comb
	 begin
	  if(opp_state_temp == 3'b000 & hcount[10:1]>=220 & hcount[10:1]<420 & vcount>=180
&vcount<480)
	    begin
		opp_on <= 1;
		addr_opp <= (hcount[10:1]-220) + (vcount-180)*200;
		end
	     else if(opp_state_temp == 3'b001 & hcount[10:1]>=215 & hcount[10:1]<415 &
vcount>=180 &vcount<480)
		begin
		opp_on <= 1;
		addr_opp <= (hcount[10:1]-215) + (vcount-180)*200;;
		end
	     else if(opp_state_temp == 3'b010 & hcount[10:1]>=225 & hcount[10:1]<425 &
vcount>=180 &vcount<480)
		begin
		opp_on <= 1;
		addr_opp <= (hcount[10:1]-225) + (vcount-180)*200;;
		end
	     else if(opp_state_temp == 3'b011 & hcount[10:1]>=180 & hcount[10:1]<380 &
vcount>=180 &vcount<480)
		begin
		opp_on <= 1;
		addr_opp <= (hcount[10:1]-180) + (vcount-180)*200;;
		end
	     else if(opp_state_temp == 3'b100 & hcount[10:1]>=260 & hcount[10:1]<460 &
vcount>=180 &vcount<480)
		begin
		opp_on <= 1;
		addr_opp <= (hcount[10:1]-260) + (vcount-180)*200;;
		end
	 else if(opp_state_temp == 3'b101 & hcount[10:1]>=140 & hcount[10:1]<340 & vcount>=180
&vcount<480)
		begin
		opp_on <= 1;
		addr_opp <= (hcount[10:1]-140) + (vcount-180)*200;;
		end
	 else if(opp_state_temp == 3'b110 & hcount[10:1]>=300 & hcount[10:1]<500 & vcount>=180
```

```
&vcount<480)
            begin
            opp_on <= 1;
            addr_opp <= (hcount[10:1]-300) + (vcount-180)*200;;
            end
        else
            begin
            opp_on <= 0;
            addr_opp <= 0;
            end
        end


     //opponent spark
   logic spark_nblank;
   assign spark_nblank = !(data_spark == 24'd16777215);

     logic spark_on;

     always_comb
        begin
         if(spark_state_temp == 2'b10 & hcount[10:1]>=275 & hcount[10:1]<395 & vcount>=170
&vcount<320)
             begin
             spark_on <= 1;
             addr_spark <= (hcount[10:1]-275) + (vcount-170)*120;
             end
           else if(spark_state_temp == 2'b11 & hcount[10:1]>=245 & hcount[10:1]<365 &
vcount>=170 &vcount<320)
             begin
             spark_on <= 1;
             addr_spark <= 119 - (hcount[10:1]-245) + (vcount-180)*120;
             end
         else
             begin
             spark_on <= 0;
             addr_spark <= 0;
             end
        end
```

```
    //opponent fist
    logic opp_fist_nblank;
    assign opp_fist_nblank = !(data_opp_fist==24'd16777215);

   logic opp_fist_on;


    always_comb
      begin
        if(opp_fist_state_temp==2'b11 & hcount[10:1]>=205 & hcount[10:1]<435 & vcount>=245
& vcount<475)
          begin
             addr_opp_fist <= 114-(hcount[10:2]-102) + (vcount[9:1]-122)*115;
              opp_fist_on <= 1;
            end
          else if(opp_fist_state_temp==2'b10 & hcount[10:1]>=263 & hcount[10:1]<378 &
vcount>=302 & vcount<417)
             begin
              addr_opp_fist <= 114-(hcount[10:1]-263) + (vcount-302)*115;
              opp_fist_on <= 1;
              end
          else
             begin
             addr_opp_fist <= 0;
              opp_fist_on <= 0;
            end
      end


    //my states
    logic atk_nblank;
    assign atk_nblank = !(data_atk == 24'd16777215);

    logic def_nblank;
    assign def_nblank = !(data_def == 24'd16777215);

    logic left_atk_on;
    logic right_atk_on;
    logic def_on;

   always_comb
      begin
          if(my_state_temp == 2'b00)//attack
             begin
```

```
                if(!isleft_temp)//right fist attack
                  begin
                     if(atk_state_temp == 3'b000 & hcount[10:1]>=378 & hcount[10:1]<503 &
vcount>=310 & vcount<480)
                       begin
                         def_on <= 0;
                         addr_def <= 0;
                         left_atk_on <= 0;
                      right_atk_on <= 1;
                      addr_atk <= (hcount[10:1]-378) + (vcount - 310)*125;
                        end
                       else if(atk_state_temp == 3'b001 & hcount[10:1]>=366 &
hcount[10:1]<491 & vcount>=300 & vcount<470)
                       begin
                         def_on <= 0;
                         addr_def <= 0;
                         left_atk_on <= 0;
                      right_atk_on <= 1;
                      addr_atk <= (hcount[10:1]-366) + (vcount - 300)*125;
                        end
                       else if(atk_state_temp == 3'b010 & hcount[10:1]>=354 &
hcount[10:1]<479 & vcount>=290 & vcount<460)
                       begin
                         def_on <= 0;
                         addr_def <= 0;
                         left_atk_on <= 0;
                      right_atk_on <= 1;
                      addr_atk <= (hcount[10:1]-354) + (vcount - 290)*125;
                        end
                       else if(atk_state_temp == 3'b011 & hcount[10:1]>=342 &
hcount[10:1]<467 & vcount>=280 & vcount<450)
                       begin
                         def_on <= 0;
                         addr_def <= 0;
                         left_atk_on <= 0;
                      right_atk_on <= 1;
                      addr_atk <= (hcount[10:1]-342) + (vcount - 280)*125;
                        end
                       else if(atk_state_temp == 3'b100 & hcount[10:1]>=330 &
hcount[10:1]<455 & vcount>=270 & vcount<440)
                       begin
                         def_on <= 0;
                         addr_def <= 0;
                         left_atk_on <= 0;
```

```
                        right_atk_on <= 1;
                        addr_atk <= (hcount[10:1]-330) + (vcount - 270)*125;
                      end
                      else if(atk_state_temp == 3'b101 & hcount[10:1]>=318 &
hcount[10:1]<443 & vcount>=260 & vcount<430)
                      begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 0;
                        right_atk_on <= 1;
                        addr_atk <= (hcount[10:1]-318) + (vcount - 260)*125;
                      end
                      else if(atk_state_temp == 3'b110 & hcount[10:1]>=306 &
hcount[10:1]<431 & vcount>=250 & vcount<420)
                      begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 0;
                        right_atk_on <= 1;
                        addr_atk <= (hcount[10:1]-306) + (vcount - 250)*125;
                      end
                    else if(atk_state_temp == 3'b111 & hcount[10:1]>=294 &
hcount[10:1]<419 & vcount>=240 & vcount<410)
                      begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 0;
                        right_atk_on <= 1;
                        addr_atk <= (hcount[10:1]-294) + (vcount - 240)*125;
                      end
                   else if(hcount[10:1]>=125 & hcount[10:1]<250 & vcount>=320 & vcount<490)

                      begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 1;
                        right_atk_on <= 0;
                        addr_atk <= (124-(hcount[10:1]-125)) + (vcount - 320)*125;
                      end
                   else
                   begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 0;
```

```
                        right_atk_on <= 0;
                        addr_atk <= 0;
                     end
                  end
              else //left fist attack
                begin
                  if(atk_state_temp == 3'b000 & hcount[10:1]>=137 & hcount[10:1]<262 &
vcount>=310 & vcount<480)
                        begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 1;
                        right_atk_on <= 0;
                        addr_atk <= 124 - (hcount[10:1]-137) + (vcount - 310)*125;
                         end
                        else if(atk_state_temp == 3'b001 & hcount[10:1]>=149 &
hcount[10:1]<274 & vcount>=300 & vcount<470)
                        begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 1;
                        right_atk_on <= 0;
                        addr_atk <= 124 - (hcount[10:1]-149) + (vcount - 300)*125;
                         end
                        else if(atk_state_temp == 3'b010 & hcount[10:1]>=161 &
hcount[10:1]<286 & vcount>=290 & vcount<460)
                        begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 1;
                        right_atk_on <= 0;
                        addr_atk <= 124 - (hcount[10:1]-161) + (vcount - 290)*125;
                         end
                        else if(atk_state_temp == 3'b011 & hcount[10:1]>=173 &
hcount[10:1]<298 & vcount>=280 & vcount<450)
                        begin
                          def_on <= 0;
                          addr_def <= 0;
                          left_atk_on <= 1;
                        right_atk_on <= 0;
                        addr_atk <= 124 - (hcount[10:1]-173) + (vcount - 280)*125;
                         end
                        else if(atk_state_temp == 3'b100 & hcount[10:1]>=185 &
hcount[10:1]<310 & vcount>=270 & vcount<440)
```

```
                            begin
                               def_on <= 0;
                               addr_def <= 0;
                               left_atk_on <= 1;
                            right_atk_on <= 0;
                            addr_atk <= 124 - (hcount[10:1]-185) + (vcount - 270)*125;
                              end
                             else if(atk_state_temp == 3'b101 & hcount[10:1]>=197 &
hcount[10:1]<322 & vcount>=260 & vcount<430)
                            begin
                               def_on <= 0;
                               addr_def <= 0;
                               left_atk_on <= 1;
                            right_atk_on <= 0;
                            addr_atk <= 124 - (hcount[10:1]-197) + (vcount - 260)*125;
                              end
                             else if(atk_state_temp == 3'b110 & hcount[10:1]>=209 &
hcount[10:1]<334 & vcount>=250 & vcount<420)
                            begin
                               def_on <= 0;
                               addr_def <= 0;
                               left_atk_on <= 1;
                            right_atk_on <= 0;
                            addr_atk <= 124 - (hcount[10:1]-209) + (vcount - 250)*125;
                              end
                           else if(atk_state_temp == 3'b111 & hcount[10:1]>=221 &
hcount[10:1]<346 & vcount>=240 & vcount<410)
                            begin
                               def_on <= 0;
                               addr_def <= 0;
                               left_atk_on <= 1;
                            right_atk_on <= 0;
                            addr_atk <= 124 - (hcount[10:1]-221) + (vcount - 240)*125;
                              end
                       else if(hcount[10:1]>=390 & hcount[10:1]<515 & vcount>=320 & vcount<490)

                            begin
                               def_on <= 0;
                               addr_def <= 0;
                               left_atk_on <= 1;
                            right_atk_on <= 0;
                            addr_atk <= (hcount[10:1]-390) + (vcount - 320)*125;
                              end
                       else
```

```
                        begin
                               def_on <= 0;
                               addr_def <= 0;
                               left_atk_on <= 0;
                           right_atk_on <= 0;
                           addr_atk <= 0;
                        end
                        end

                  end

              else if(my_state_temp == 2'b01)//defend
                begin
                  if(def_state_temp==1)//not hurt
                  begin
                  if(hcount[10:1]>=190 & hcount[10:1]<315 & vcount>=310 & vcount<480)
                begin
                   def_on <= 1;
                     addr_def <= (124-(hcount[10:1]-190)) + (vcount-310)*125;
                     left_atk_on <= 0;
                     right_atk_on <= 0;
                     addr_atk <= 0;
                  end
                else if(hcount[10:1]>=325 & hcount[10:1]<450 & vcount>=310 & vcount<480)
                begin
                   def_on <= 1;
                   addr_def <= (hcount[10:1]-325) + (vcount-310)*125;
                     left_atk_on <= 0;
                     right_atk_on <= 0;
                     addr_atk <= 0;
                   end
                else
                begin
                   def_on <= 0;
                   addr_def <= 0;
                     left_atk_on <= 0;
                     right_atk_on <= 0;
                     addr_atk <= 0;
                   end
                  end
                  else//hurt
                  begin
                  if(hcount[10:1]>=190 & hcount[10:1]<315 & vcount>=320 & vcount<490)
                   begin
```

```
                    def_on <= 1;
                     addr_def <= (124-(hcount[10:1]-190)) + (vcount-320)*125;
                     left_atk_on <= 0;
                     right_atk_on <= 0;
                     addr_atk <= 0;
                     end
                 else if(hcount[10:1]>=325 & hcount[10:1]<450 & vcount>=320 & vcount<490)
                    begin
                    def_on <= 1;
                    addr_def <= (hcount[10:1]-325) + (vcount-320)*125;
                     left_atk_on <= 0;
                     right_atk_on <= 0;
                     addr_atk <= 0;
                    end
                 else
                    begin
                    def_on <= 0;
                    addr_def <= 0;
                     left_atk_on <= 0;
                     right_atk_on <= 0;
                     addr_atk <= 0;
                    end
                   end

                 end


        else if(my_state_temp == 2'b10)//idle
                 begin
                  if(idle_state_temp == 2'b00)
                           begin
                             if(hcount[10:1]>=390 & hcount[10:1]<515 & vcount>=320 & vcount<490)
                            begin
                               def_on <= 0;
                               addr_def <= 0;
                               left_atk_on <= 0;
                           right_atk_on <= 1;
                           addr_atk <= (hcount[10:1]-390) + (vcount - 320)*125;
                              end
                       else if(hcount[10:1]>=125 & hcount[10:1]<250 & vcount>=320 & vcount<490)

                            begin
                               def_on <= 0;
                               addr_def <= 0;
```

```
left_atk_on <= 1;
right_atk_on <= 0;
addr_atk <= (124-(hcount[10:1]-125)) + (vcount - 320)*125;
end
else
begin
def_on <= 0;
addr_def <= 0;
left_atk_on <= 0;
right_atk_on <= 0;
addr_atk <= 0;
end
end

else if(idle_state_temp == 2'b01)
begin
if(hcount[10:1]>=385 & hcount[10:1]<510 & vcount>=320 & vcount<490)
begin
def_on <= 0;
addr_def <= 0;
left_atk_on <= 0;
right_atk_on <= 1;
addr_atk <= (hcount[10:1]-385) + (vcount - 320)*125;
end
else if(hcount[10:1]>=120 & hcount[10:1]<245 & vcount>=320 & vcount<490)

begin
def_on <= 0;
addr_def <= 0;
left_atk_on <= 1;
right_atk_on <= 0;
addr_atk <= (124-(hcount[10:1]-120)) + (vcount - 320)*125;
end
else
begin
def_on <= 0;
addr_def <= 0;
left_atk_on <= 0;
right_atk_on <= 0;
addr_atk <= 0;
end
end

else if(idle_state_temp == 2'b10)
```

```
                    begin
                        if(hcount[10:1]>=395 & hcount[10:1]<520 & vcount>=320 & vcount<490)
                        begin
                            def_on <= 0;
                            addr_def <= 0;
                            left_atk_on <= 0;
                        right_atk_on <= 1;
                        addr_atk <= (hcount[10:1]-395) + (vcount - 320)*125;
                        end
                    else if(hcount[10:1]>=130 & hcount[10:1]<255 & vcount>=320 & vcount<490)

                        begin
                            def_on <= 0;
                            addr_def <= 0;
                            left_atk_on <= 1;
                        right_atk_on <= 0;
                        addr_atk <= (124-(hcount[10:1]-130)) + (vcount - 320)*125;
                        end
                        else
                        begin
                        def_on <= 0;
                    addr_def <= 0;
                    left_atk_on <= 0;
                    right_atk_on <= 0;
                    addr_atk <= 0;
                        end
                end

                else
                        begin
                        def_on <= 0;
                    addr_def <= 0;
                    left_atk_on <= 0;
                    right_atk_on <= 0;
                    addr_atk <= 0;
                        end
            end

        else //hurt
            begin
                def_on <= 0;
                addr_def <= 0;
                left_atk_on <= 0;
                right_atk_on <= 0;
```

```
                    addr_atk <= 0;
              end

          end


    /*******************************************************************************
*******/


    /*********************************priority
controller*********************************/
    always_comb
    begin
        if(start_on)
            {VGA_R, VGA_G, VGA_B} = data_start * 24'd16777215;

        else if(title_on)
            {VGA_R, VGA_G, VGA_B} = data_title * 24'd16777215;

            else if(num1_on & num1_nblank)
            {VGA_R, VGA_G, VGA_B} = data_num1 * 24'd8496056;

        else if(num2_on & num2_nblank)
            {VGA_R, VGA_G, VGA_B} = data_num2 * 24'd8496056;

        else if(num3_on & num3_nblank)
            {VGA_R, VGA_G, VGA_B} = data_num3 * 24'd8496056;


        else if(my_state_temp == 2'b11 & opp_fist_on & opp_fist_nblank)
            {VGA_R, VGA_G, VGA_B} = data_opp_fist;

            else if((left_atk_on | right_atk_on) & atk_nblank)
            {VGA_R, VGA_G, VGA_B} = data_atk;

            else if(def_on & def_nblank)
            {VGA_R, VGA_G, VGA_B} = data_def;

            else if(opp_fist_on & opp_fist_nblank)
      {VGA_R, VGA_G, VGA_B} = data_opp_fist;

            else if(spark_on & spark_nblank)
```

```
          {VGA_R, VGA_G, VGA_B} = data_spark;

         else if(opp_on & opp_nblank)
          {VGA_R, VGA_G, VGA_B} = data_opp;

         else if(border_on & (!title_state_temp))
           {VGA_R, VGA_G, VGA_B} = 24'd0;

         else if(myhealth_on & (!title_state_temp))
           {VGA_R, VGA_G, VGA_B} = {8'hff,8'h0,8'h0};

         else if(opphealth_on & (!title_state_temp))
           {VGA_R, VGA_G, VGA_B} = {8'h0,8'h0,8'hff};

         else if(bg_up_on& (!title_state_temp))
          {VGA_R, VGA_G, VGA_B} = 24'd16777215;

         else if(bg_mid_on& (!title_state_temp))
           {VGA_R, VGA_G, VGA_B} = data_background;

         else if(bg_btm_on& (!title_state_temp))
           {VGA_R, VGA_G, VGA_B} = 24'd8496056;


         else
           {VGA_R, VGA_G, VGA_B} = 24'd16777215;;



     end


/*****************************************************************************************
***/
endmodule // VGA_LED_Emulator
```

9.2.3 Audio_effects.sv

```
//Original audio codec code taken from
//Howard Mao's FPGA blog
//http://zhehaomao.com/blog/fpga/2014/01/15/sockit-8.html
//MOdified as needed
```

```
/* audio_effects.sv
    Reads the audio data from the ROM blocks and sends them to the
    audio codec interface
*/

module audio_effects (
    input  clk, //audio clock
    input  sample_end, //sample ends
    input  sample_req, //request new sample
    output [15:0] audio_output, //sends audio sample to audio codec
  //  input [15:0] M_bell,    //bell sound ROM data
    input [15:0] M_dfd,    //city sound ROM data
    input [15:0] M_ddg, //whoosh sound ROM data
    input [15:0] M_hrt,  //sword sound ROM data
  //  output [14:0] addr_bell,    //ROM addresses
    output [13:0] addr_ddg,
    output [12:0] addr_dfd,
    output [13:0] addr_hrt,
    input  [1:0] control,    //Control from avalon bus
      input  [2:0] ctrl
);


reg  [13:0]  index_hrt = 14'd0;     //index through the sound ROM data for different sounds
reg  [13:0]  index_ddg = 14'd0;
reg  [12:0]  index_dfd = 13'd0;
//reg  [15:0]  index_sw = 15'd0;
//reg [15:0] count1 = 15'd0;
//reg [15:0] count2 = 15'd0;



reg [15:0] dat;

assign audio_output = dat;

//assign index to ROM addresses
always @(posedge clk) begin
    addr_hrt <= index_hrt;
    addr_ddg <= index_ddg;
    addr_dfd <= index_dfd;
 //   addr_who <= index_who;
 //   addr_sw <= index_sw;

end
```

```
//Keep playing background (city) sound if control is off
//Play sword sound if control is ON

always @(posedge clk)
begin

    if (sample_req)
       begin
      if(control == 2'b00)
         begin
         dat <= M_hrt;
             if(index_hrt == 14'd9514)
                index_hrt <= 14'd0;
             else
                  index_hrt <= index_hrt + 1'b1;
             end
           else if(control == 2'b01)
             begin
             dat <= M_ddg;
             if(index_ddg == 14'd8999)
                index_ddg <= 14'd0;
             else
                index_ddg <= index_ddg + 1'b1;
             end
           else if(control == 2'b10)
             begin
             dat <= M_dfd;
             if(index_dfd == 13'd7999)
                index_dfd <= 13'd0;
             else
                index_dfd <= index_dfd + 1'b1;
             end
            else
                dat <= 16'd0;
           end

       else
           dat <= 16'd0;
end

endmodule
```

9.2.4 Audio_codec.sv

```
// Original audio codec code taken from
//Howard Mao's FPGA blog
//http://zhehaomao.com/blog/fpga/2014/01/15/sockit-8.html
//MOdified as needed

/*
audio_codec.sv
Sends samples to the audio codec ssm 2603 at audio clock rate.
*/



//Audio codec interface
module audio_codec (
    input  clk, //audio clock
    input  reset,
    output [1:0]  sample_end,   //end of sample
    output [1:0]  sample_req,   //request new sample
    input  [15:0] audio_output, //audio output sent to audio codec
    input  [1:0] channel_sel,   //select channel
    output AUD_ADCLRCK, //ADC channel clock
    input AUD_ADCDAT,
    output AUD_DACLRCK, //DAC channel clock
    output AUD_DACDAT,
    output AUD_BCLK //Bit clock
);

// divided by 256 clock for the LRC clock, one clock is oen audio frame
reg [7:0] lrck_divider;

// divided by 4 clock for the bit clock BCLK
reg [1:0] bclk_divider;

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

wire lrck = !lrck_divider[7];

//assigning clocks from the clock divider
assign AUD_ADCLRCK = lrck;
assign AUD_DACLRCK = lrck;
assign AUD_BCLK = bclk_divider[1];
```

```
// assigning data as last bit of shift register
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

//first 16 bit sample sent after 16 bclks or 4*16=64 mclk
assign sample_end[1] = (lrck_divider == 8'h40);
//second 16 bit sample sent after 48 bclks or 4*48 = 192 mclk
assign sample_end[0] = (lrck_divider == 8'hc0);

// end of one lrc clk cycle (254 mclk cycles)
assign sample_req[1] = (lrck_divider == 8'hfe);
// end of half lrc clk cycle (126 mclk cycles) so request for next sample
assign sample_req[0] = (lrck_divider == 8'h7e);

wire clr_lrck = (lrck_divider == 8'h7f); // 127 mclk
wire set_lrck = (lrck_divider == 8'hff); // 255 mclk
// 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]);

//implementing shift operation to send the audio samples
always @(posedge clk)
begin
    if (reset)
       begin
        shift_out <= 16'h0;
        shift_temp <= 16'h0;
       end
     else if (set_lrck)
       begin
            shift_out <= audio_output;
            shift_temp <= audio_output;
      end
```

```
    else if (clr_lrck)
      begin
            shift_out <= shift_temp;
      end
    else if (clr_bclk == 1)
      begin
        shift_out <= {shift_out[14:0], 1'b0};
      end
end

endmodule
```

9.2.5 Audio_Top.sv

```
// Original audio codec code taken from
//Howard Mao's FPGA blog
//http://zhehaomao.com/blog/fpga/2014/01/15/sockit-8.html
//MOdified as needed

/* Audio_top.sv
Contains the top-level audio controller. Instantiates sprite ROM blocks and
communicates with the avalon bus */

module Audio_top (
    input  OSC_50_B8A,    //reference clock
    input  [1:0] audio_ctrl,
     inout  AUD_ADCLRCK, //Channel clock for ADC
    input  AUD_ADCDAT,
    inout  AUD_DACLRCK, //Channel clock for DAC
    output AUD_DACDAT,  //DAC data
    output AUD_XCK,
    inout  AUD_BCLK, // Bit clock
    output AUD_I2C_SCLK, //I2C clock
    inout  AUD_I2C_SDAT, //I2C data
    output AUD_MUTE,    //Audio mute
    input  [3:0] KEY,
    input  [3:0] SW
 //   output [3:0] LED
);

wire reset;
wire main_clk;
```

```
wire audio_clk;

assign reset = !KEY[0];

//reg ctrl;
//wire chipselect = 1;

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


//Sound samples from audio ROM blocks
wire [15:0] M_hrt;
//wire [15:0] M_bgm;
wire [15:0] M_dfd;
wire [15:0] M_ddg;



//Audio ROM block addresses
wire [13:0] addr_hrt;
wire [13:0] addr_ddg;
wire [12:0] addr_dfd;
//wire [16:0] addr_bgm;



//Store sounds in memory ROM blocks
hrt hrt(.clock(OSC_50_B8A), .address(addr_hrt), .q(M_hrt));
ddg ddg(.clock(OSC_50_B8A), .address(addr_ddg), .q(M_ddg));
dfd dfd(.clock(OSC_50_B8A), .address(addr_dfd), .q(M_dfd));


//generate audio clock
clock_pll pll (
    .refclk (OSC_50_B8A),
    .rst (reset),
    .outclk_0 (audio_clk),
    .outclk_1 (main_clk)
);

//Configure registers of audio codec ssm2603
i2c_av_config av_config (
```

```
    .clk (audio_clk),
    .reset (reset),
    .i2c_sclk (AUD_I2C_SCLK),
    .i2c_sdat (AUD_I2C_SDAT),
    .status (LED_FAKE)
);

logic [3:0] LED_FAKE;

assign AUD_XCK = audio_clk;
assign AUD_MUTE = (SW != 4'b0);



//Call Audio codec interface
audio_codec ac (
    .clk (audio_clk),
    .reset (reset),
    .sample_end (sample_end),
    .sample_req (sample_req),
    .audio_output (audio_output),
    .channel_sel (2'b10),

    .AUD_ADCLRCK (AUD_ADCLRCK),
    .AUD_ADCDAT (AUD_ADCDAT),
    .AUD_DACLRCK (AUD_DACLRCK),
    .AUD_DACDAT (AUD_DACDAT),
    .AUD_BCLK (AUD_BCLK)
);

//Fetch audio samples from these ROM blocks
audio_effects ae (
    .clk (audio_clk),
    .sample_end (sample_end[1]),
    .sample_req (sample_req[1]),
    .audio_output (audio_output),
  //  .addr_bell(addr_bell),
    .addr_dfd(addr_dfd),
    .addr_ddg(addr_ddg),
    .addr_hrt(addr_hrt),
    .M_dfd(M_dfd),
    .M_ddg(M_ddg),
    .M_hrt(M_hrt),
    .control(audio_ctrl),
```

```
    .ctrl(KEY[3:1])
);

Endmodule
```

9.2.6 i2c_av_config.sv

```
// Original audio codec code taken from
//Howard Mao's FPGA blog
//http://zhehaomao.com/blog/fpga/2014/01/15/sockit-8.html
//MOdified as needed

//configure Audio codec using the I2C protocol
module i2c_av_config (
    input clk,
    input reset,

    output i2c_sclk, //I2C clock
    inout  i2c_sdat,   // I2C data out

    output [3:0] status
);

reg [23:0] i2c_data;
reg [15:0] lut_data;
reg [3:0]  lut_index = 4'd0;

parameter LAST_INDEX = 4'ha;

reg  i2c_start = 1'b0;
wire i2c_done;
wire i2c_ack;

//Send data to I2C controller
i2c_controller control (
    .clk (clk),
    .i2c_sclk (i2c_sclk),
    .i2c_sdat (i2c_sdat),
    .i2c_data (i2c_data),
    .start (i2c_start),
    .done (i2c_done),
    .ack (i2c_ack)
```

```
);

//configure various registers of audio codec ssm 2603
always @(*) begin
    case (lut_index)
        4'h0: lut_data <= 16'h0c10; // power on everything except out
        4'h1: lut_data <= 16'h0017; // left input
        4'h2: lut_data <= 16'h0217; // right input
        4'h3: lut_data <= 16'h0479; // left output
        4'h4: lut_data <= 16'h0679; // right output
        4'h5: lut_data <= 16'h08d4; // analog path
        4'h6: lut_data <= 16'h0a04; // digital path
        4'h7: lut_data <= 16'h0e01; // digital IF
        4'h8: lut_data <= 16'h102c; // sampling rate
        4'h9: lut_data <= 16'h0c00; // power on everything
        4'ha: lut_data <= 16'h1201; // activate
        default: lut_data <= 16'h0000;
    endcase
end

reg [1:0] control_state = 2'b00;

assign status = lut_index;

always @(posedge clk) begin
    if (reset) begin
        lut_index <= 4'd0;
        i2c_start <= 1'b0;
        control_state <= 2'b00;
    end else begin
        case (control_state)
            2'b00: begin
                i2c_start <= 1'b1;
                i2c_data <= {8'h34, lut_data};
                control_state <= 2'b01;
            end
            2'b01: begin
                i2c_start <= 1'b0;
                control_state <= 2'b10;
            end
            2'b10: if (i2c_done) begin
                if (i2c_ack) begin
                    if (lut_index == LAST_INDEX)
                        control_state <= 2'b11;
```

```
                    else begin
                        lut_index <= lut_index + 1'b1;
                        control_state <= 2'b00;
                    end
                end else
                    control_state <= 2'b00;
            end
        endcase
    end
end

endmodule
```