NUNY: Ninja University in the City of New York
Final Report

CSEE 4840 Embedded System Design

Kshitij Bhardwaj, Van Bui, Vinti Vinti, and Kuangya Zhai

{kb2673, vb2363, vv2236, kz2219}@columbia.edu

May 14, 2014

Contents

1 Overview

In this project, we design and implement a Fruit Ninja like video game on the Arrow SoCKit development board [1]. Fruit Ninja is a popular video game where the player slices fruit with their finger(s) on a touch screen. The theme of our game will be based on undergraduate/graduate school life so that rather than slicing fruit, the object of the game will be to slice assignments, exams, thesis writing, food (like pizza), and books. The game will generate several moving objects on the screen and the player will destroy objects using an on screen ninja with a sword controlled by a wiimote controller.

NUNY has three levels to the game representing each stage of higher education (i.e. bachelors, masters, and doctorate). Each stage varies in level of difficulty, with the doctorate being the toughest to complete. The ninja student will have to earn a minimum score and have lives remaining (out of three) to pass each stage. There will be several objects appearing and disappearing from the screen and the player will have to slice certain objects in order to increase their score. There will also be objects that the player should not slice, such as the letter F, as it will cause them to lose one life. The player must slice a valid object in time before it disappears from the screen in order to obtain points, otherwise they will lose one life for each object that they do not slice in time. The entire game is won when the player completes their doctorate degree successfully.

Figure 1 is a snapshop of the start screen of the NUNY video game. The score and lives can be seen at the very top of the screen and at the bottom of the screen are the three levels to the game that can be selected. The ninja student at the center will slice the objects flying around the screen once a level is selected by the player.

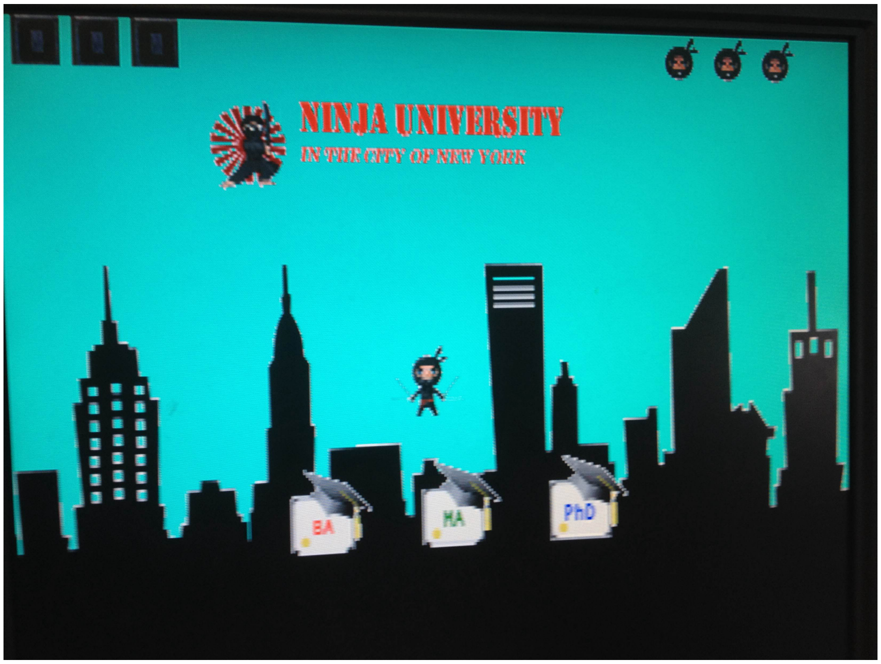

Figure 1: Start screen for NUNY game.

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 controller as well as the device drivers in order to control the audio and movement of sprites. 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, both stationary and moving, for the background, moving objects, scores, etc. 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.

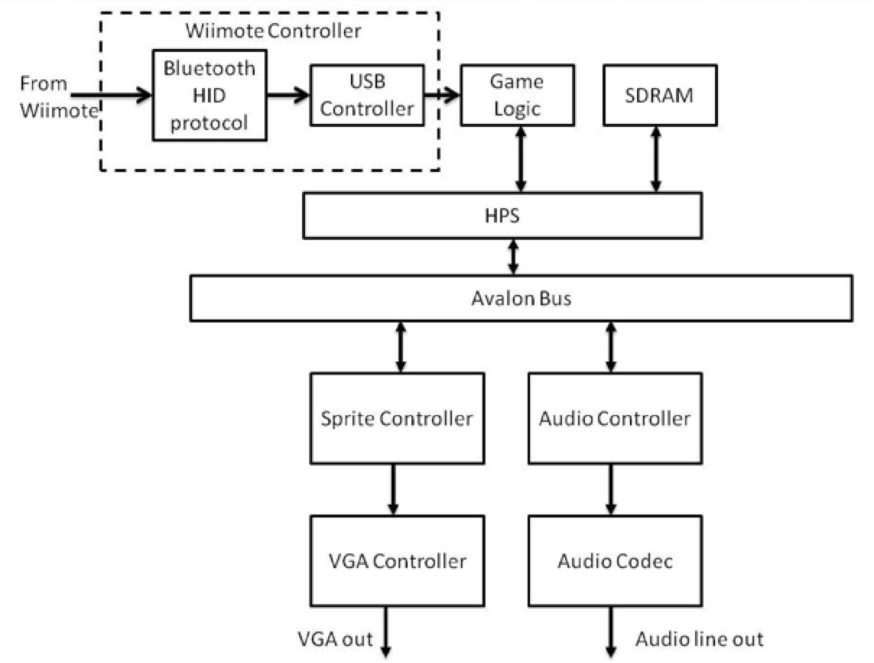

Figure 2: High level software and hardware design components.

3 Graphics and Audio Preparation

The preparations required for the graphics and audio are similar. First the image and audio files had to be searched for online. Once we agreed on the images and audio for the game, we edited them to fit our game design. Finally, both the image and audio files had to be converted to MIF format in order to be stored in the on-chip ROM blocks.

3.1 Audio Preparation

For the audio, we include audio for the background music and also sound effect. We decided to use the Ogg Vorbis format and were able to find several audio files online in this format. For each of the audio files, we edited the audio files for length, channels, and sampling rate using the sox utility. To save on memory space, we shorten the audio file length to play for about one second each. The audio is configured with a sampling rate of 44100 Hz and 16 bit quantization for good quality.

Similar to what we did with the images, we also converted the audio files in Ogg to the MIF format in order to store them in on-chip ROM blocks. We modified some C++ code we found online to partially convert the files to MIF. The code uses the Ogg Vorbis SDK to decode the Ogg files and reads the relevant information about the compressed audio data. We modified the code to C and to also output 16-bit samples using the MIF format.

For overlapping sounds, we tested both adding and averaging the audio samples. We found that adding the audio samples of overlapping sounds provided the best quality. Despite this, we ended up not overlapping the sounds since we found that not overlapping the sounds with just two audio files also provided good sound quality while debugging another issue with the audio implementation.

The two audio sounds we have is the background and the sword sound effect. The background audio is the sound of city drums. The amount of ROM space required for the background audio is about 44 KB. The background audio plays continuously throughout the game and so does not require any software controls. In contrast, the sword sound effect is controlled by software since it only plays when the ninja successfully slices an object. The sword sound effect uses about 33 KB of on-block ROM. So in total, the audio files utilized about 77 KB of on-chip ROM.

3.2 Graphics Preparation

First, we gathered several images (30+) via the web and edited the images to match our game design. Our game includes images for both stationary and moving objects. For example, the student ninja will be moving around as well as his sword, while objects like the New York City skyline are stationary (see Figure 3). NUNY includes images for the scores, lives, ninja student, the current weather, objects to slice, level selection, try again option, diploma, the NYC skyline, and pass/fail. All of our images were 64x64 pixel images with the exception of the NYC skyline, which was 200x160, and the numbers and lives, which were both 32x32.

Figure 3: Examples of stationary and moving sprites.

The image files we collected varied in different image file formats and we needed the image files to be in the MIF (memory initialization format) format since they will eventually be stored on the on-chip ROM blocks. We found a code written in matlab online that we modified to translate our image files into the mif format, the original code created COE files. The matlab code also resized our images. Table 1 lists each image and their sizes. The total amount of memory for the graphics was about 400 KB.

Block	**Number of Sprites**	**Pixel Size**	**Total ROM Size(bytes)**
Numbers	10	32x32	61440
Lives	1	32x32	1536
Ninja	3	64x64	18432
Weather	3	64x64	18432
Slicing Objects	6	64x64	36864
Level Selection	3	64x64	18432
Try Again	1	64x64	6144
Diploma	1	64x64	6144
NYC Skyline	4	200x160	192000
Pass/Fail	2	64x64	96000
Total	34	−−	449280

Table 1: Graphics Memory Budget

4 Wii Controller

There are three devices needed for the Wiimote Controller model: (i) Wiimote, (ii) Bluetooth USB Dongle, and (iii) Sensor Bar. The sensor bar emits infrared signal when powered and should be placed in front of the screen. The Bluetooth dongle connects to the SoCKit board through the USB interface and standard Bluetooth HID protocol, and receives Bluetooth signal sending from the Wiimote controller. There are two sensors built in the Wiimote: the accelerometer and the front digital camera. The accelerometer senses the acceleration of the Wiimote and the front digital camera senses the relative position of the Wiimote to the sensor bar. The Wiimote then sends the acceleration and position information to the SoCKit board through the Bluetooth USB dongle.

We use BlueZ [2] as the Bluetooth stack to communicate between the Wiimote and Linux host. libwiimote [3] is a C-library build on BlueZ that provides a simple API for communicating between the Wiimote and the Linux host. We can get the data of the accelerometer and ir-sensor 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.

5 Game Logic Controller

Game logic is implemented in software using C programming language. The key functions of the game logic controller are to control the generation of sprites (graphics), read location of Wii pointer through the Wiimote controller, generate appropriate audio when required during the game by interacting with the audio controller and finally implement the actual game logic, its rules and compute the playerś GPA, based on how many program requirements heshe has fulfilled (or sliced). Each of the above functions are implemented as a submodule of the game logic controller. As shown in Figure 4, there are 4 submodules, which are described in more detail next.

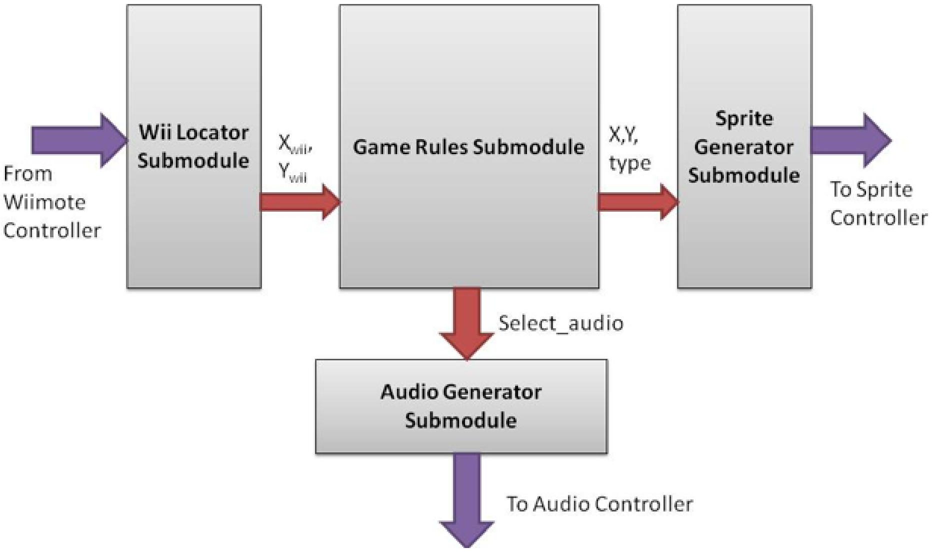

Figure 4: Game Logic Controller block diagram.

1. **Game rules:** This is the main submodule of the game logic controller and it interfaces with all the other submodules, instructing them what to do and when based on the rules of the game. For example, to create the screen where a player selects a program (Phd, MS, or undergrad), the game rules submodule tells the sprite generator submodule to generate sprites such as "MS" / "PhD" etc. It also tells the audio generator to interact with the audio controller to generate the background music for this opening screen.

This submodule is responsible for the dynamic behavior of the game and keeps updating the screen according to the game being played. It also implements the logic to determine the speed of the various sprites on the screen. It calculates the final GPA of the user by mapping the no. of program requirement sprites he/she has sliced to an actual GPA score for the whole semester.

2. **Sprite Generator:** Based on the game logic, this submodule generates the X and Y coordinates of the different sprites that need to be displayed on the screen. These X and Y coordinates for each sprite are stored in memory (using iowrite calls), which gets updated according to the actual game logic. This memory is accessed by the sprite controller through the address bits, which then displays the necessary sprites on the screen.

   The X and Y coordinates for the moving sprites will be determined based on the current time step, velocity in the x and y direction, gravity, and the initial x and y coordinate positions.

   $x(t) = x\_velocity \times time + x\_init$
   $y(t) = -\frac{1}{2} \times gravity \times time^2 + y\_velocity \times time + y\_init$

3. **Wii Locator:** Game logic controller interacts with the Wii controller to determine the location where the Wiimote is pointing. The Wii locator submodule also interacts with the game rules submodule (which then talks to the sprite generator) to select the appropriate sprite based on the Wii location. For example, if the X,Y coordinates obtained from the Wii controller (which are the coordinates of the sword) are within the dimensions of a sprite (say the homework sprite) then the homework sprite needs to be updated to a new sprite which shows a sliced homework. A more simpler example will be the movement of sword, which is displaying a sword sprite at the exact position where the Wii is pointing.

4. **Audio Generator:** Various audio sounds that need to be generated throughout the game (background music, slicing sounds, etc.) are encoded inside the audio generator submodule. Based on the game logic, this submodule tells the audio controller to generate the appropriate sound while the game is being played. For example, if the player successfully completes a level, the game logic will tell the audio controller to play the graduation music.

6 Device Drivers

6.1 Audio Device Driver

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 4-bit address bits to access 16 locations that store 16-bit data. The data written to peripheral memory-mapped device using the device driver include the x, y positions of the moving sprites. In addition to ninja, there are 5 other moving sprites. Together these sprites occupy 12 of the 16 locations. The remaining 4 locations are occupied by the scores, remaining lives, selecting the screen and the levels. Both the positions and other statistics are written using the ioctl calls from the game logic module during each timestamp.

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.

6.2 Graphics Device Driver

The audio device driver writes the control bit using the write ioctl call. This control bit is used to switch on/off the slicing sound in the hardware. The memory-mapped audio peripheral has 1-bit address and stores the control in 16-bit data. The ioctl calls are made from the game logic whenever the ninja intersects any moving sprite.

Similar to the VGA driver, there are extensive bound checking and only write ioctl calls are made.

Both the audio and VGA peripherals, with their base addresses are added in the dts file, which is then compiled to generate the device tree blob.

7 Sprite Controller and VGA Display

The video display controller has two submodules, the VGA Controller(from lab3) and the Sprite Controller(RGB Controller) (see Figure 5). The Sprite controller has been implemented using three line buffers; two line buffers to write into at alternate rows and one line buffer to read from continuously. Figure 6 gives the detailed top level interconnections between the modules used in the VGA_LED which is the top level design for the display module.

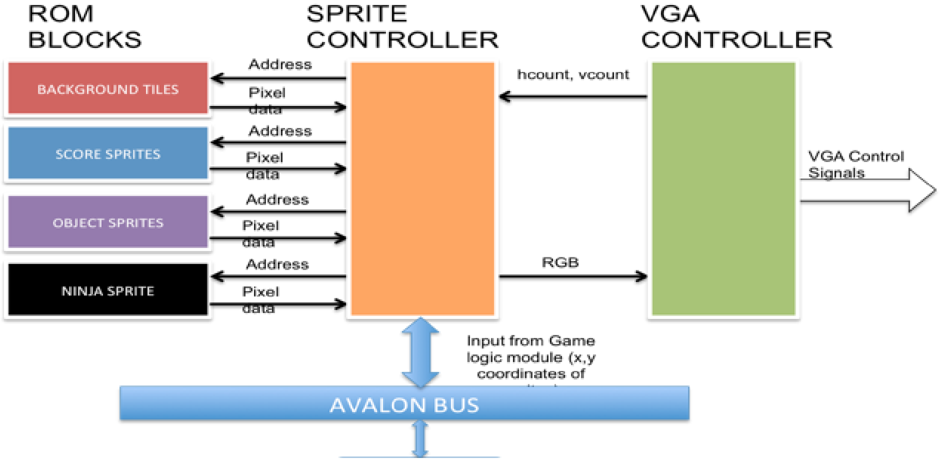

Figure 5: Video display controller block diagram.

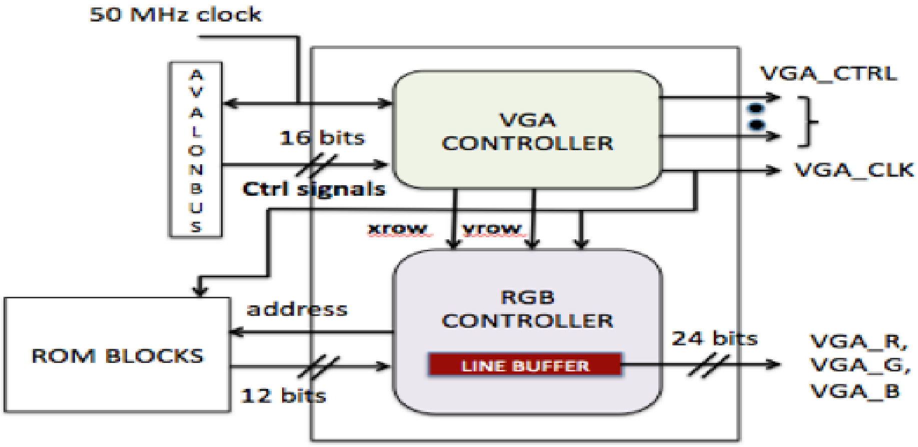

Figure 6: Video display controller top level.

1. **VGA Controller:** This module generates the VGA signals and also the hcount and vcount values that is used in the RGB Controller module to locate x and y coordinates of the VGA display. All the control signals for VGA except for VGA_R, VGA_G, VGA_B are generated in this module. VGA_CLK (25MHz) is used as the clock for the RGB controller.

2. **RGB Controller:** The sprite controller sends the RGB values of each pixel (depending on the current hcount value) to the VGA Controller. These values are read from a 640(24 bit word) line buffer. The inputs for the controller are the following:

   - Hcount and Vcount (current position of the pixel)

- X and Y coordinates of the Ninja
- X and Y coordinates of the object sprites
- Screen selection bits
- Level selection bits
- current scores
- current life left

The game consists of 4 layers (see Figure 7). The order of the layers is as follows:

- The background layer has the lowest priority
- The score display layer comes next
- The object layer is next and has 6 to 8 sub layers, depending on the difficulty level of the game
- The topmost layer is the ninja and it has the highest priority

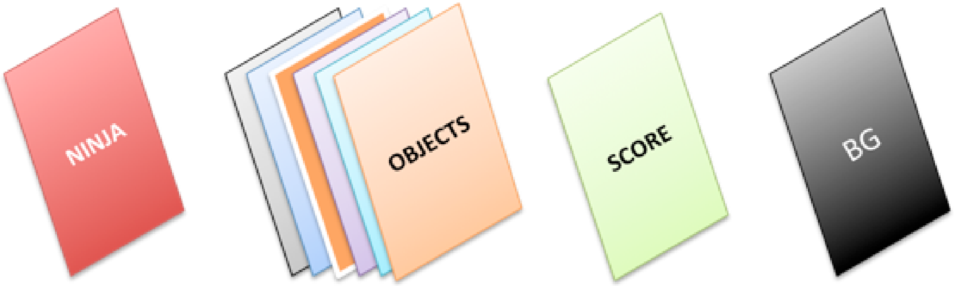

Figure 7: VGA Display Layers.

The sprite controller submodule gets the coordinate inputs and screen/level selection inputs from the game logic controller through the avalon bus, specifying the position of the sprites on the screen. It has two line buffers of size 640 (for each pixel in a line of the VGA screen). At a given time, it will write the value of each pixel in one line buffer and read out the other line buffer to the VGA line buffer. The read and write operations are done at a clock frequency of 25MHz i.e. the VGA_CLK.

3. **Line Buffer Write Operation:** The write operation in the line buffers can be summarized in the following points.

   The write operation in the line buffers can be summarized in the following points.

   - All the control signals for the sprites display and position have been derived from the signals coming from the game logic through avalon bus.
   - There are two line buffers (640x1, each word 12 bits) that are being written into, one row at a time and a third line buffer than simply copies the data from the previous line buffer that was written into. The RGB pixel information is read from this third line buffer. Refer to Figure 8 for this operation.
   - In order to simplify the design and implementation, the address calculation, data fetch, and pixel selection (using a priority encoder) have been done in parallel for each layer (background, score, lifes, objects, ninja) and in combinational logic. Only the write operation into the buffer is clocked (at 25MHz). This approach solved the timing issues that was earlier being encountered when using sequential logic at every stage (see Figure 9).
   - A 1 bit counter(cnt), that counts the value of xrow (derived from hcount) is used to select which of the two buffers to write into and to read from. Hence at a given hcount value, if the write is being done to linebuffer1, the read is done from linebuffer2.

4. **Memory Budget for sprites:** Each pixel is represented using 12 bits (4 bits each for RGB). See Table 1 for sprite memory usage details.

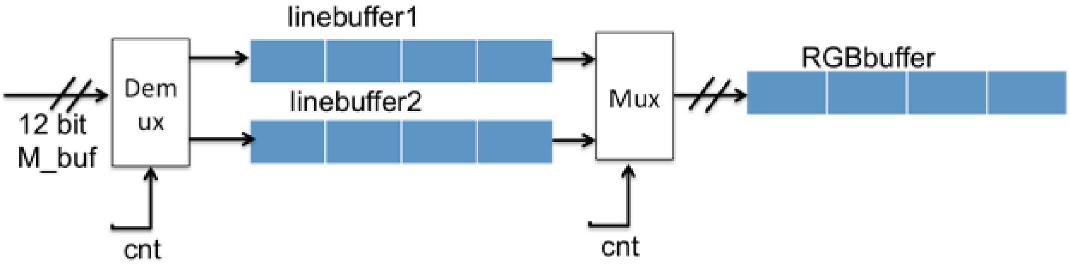

Figure 8: Line buffer write operation.

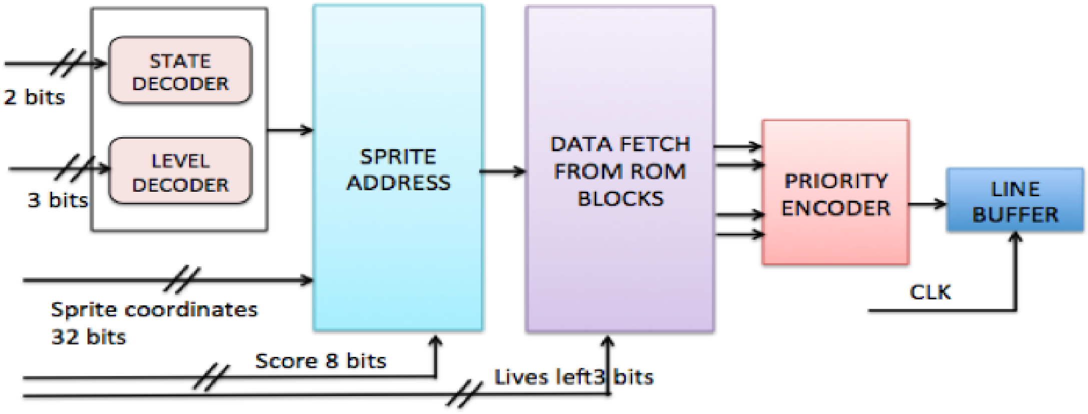

Figure 9: Sprite controller operation.

8 Audio Controller

The SoCKit board supports 24-bit audio with the Analog Devices SSM2603 audio codec. SSM2603 has ports for microphone in, line in, and line out. The sampling rate supported is 8 KHz to 96 KHz and is adjustable.

NUNY supports sound for object slicing and background music. The audio controller has 3 main components: 1) Audio Data, 2) Audio codec configuration interface 3) Digital audio interface. The complete block diagram is shown in Figure 10. These components are described in more detail below.

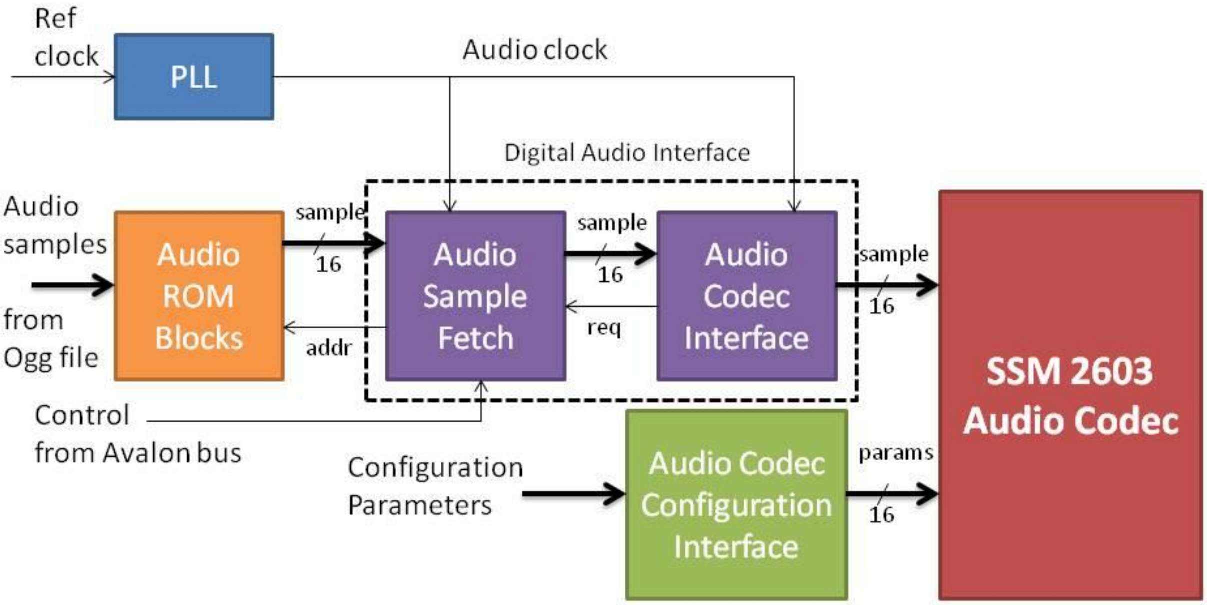

Figure 10: The block diagram of the audio controller.

Audio Data The two sound files are converted from ogg format to mif format. These mif files for the background sound (city.mif) and the slicing sound (sword.mif) are used to create ROM data blocks using megawizard. Background music ROM block contains 22049 16-bit audio samples and slicing sound ROM block contains 16537 16-bit audio samples. The total size of the memory used for audio storage is 77KB.

Audio Codec Configuration Interface This interface is used to configure the various parameters inside the SSM 2603 audio codec. This interface uses the I2C protocol to communicate the configuration parameters to the audio codec. Some of the configured parameters are: volume (which is set to 0 db), the mode of the audio codec (which is set to slave), sampling rate (we are using 44.1 kHz), power on and off the audio codec, etc.

Digital Audio Interface This interface has two sub-components: a) Audio sample fetch and b) Audio codec interface. Both of these sub-components operate at the audio clock rate (11.3 Mhz), which is derived from the reference clock (50 Mhz) using Phase Locked Loop (PLL).

The audio sample fetch is used to get the 16-bit audio samples from the Audio ROM blocks, which are accessed using the address bits for the blocks. The fetch unit also takes control as input, which comes from the audio peripheral module in software. This control signal is used to control the switching on and off of the slicing sound.

The Audio codec interface sub-component sends audio samples to the audio codec using shift registers, that shift these samples at fixed clock rate. The audio clock is used to derive two audio clocks: (i) Left Right Channel (LRC) clock and (ii) Bit clock. Both these clocks are generated from the audio clock using clock divider.

The LRC clock is used for time multiplexing the audio samples. The audio sample can be sent out on the positive phase (left channel) of the clock or negative phase (right channel). The bit clock is used to send each bit of the audio sample as shown by the timing diagram in Figure 11. Please note as there are many number of cycles in one phase of the LRC clock, the codec interface sends don't cares for the remaining cycles are after transmitting 16 bits of the audio sample.

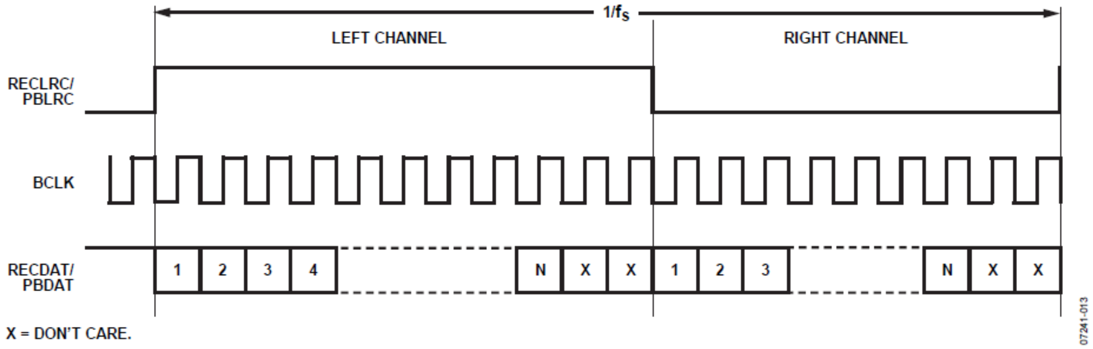

Figure 11: The timing diagram of the audio sample.

9 Experience and Issues

One of the biggest issue that we experienced was setting up of the Wiimote connection. This involved recompiling the kernel and enabling the bluetooth device to get our bluetooth dongle working. Even then, we observed that the Wiimote is unable to connect when we connect the bluetooth dongle directly to the FPGA's USB port. We later realized that we have to use the USB hub to connect the dongle. But we spent many weeks trying to debug this.

Due to limited on-chip FPGA memory, we decided to store our audio files on the HPS memory. For this we implemented an audio buffer that sent interrupts to the software when it needed more audio samples. We also got our interrupt device driver working and tried playing the sound. But the sound quality was poor and we finally gave up on the idea and decided to store the sound also on the FPGA memory. Fortunately, the FPGA memory was enough to store both audio and sprites.

We also simplified testing and debugging of our hardware code using very modular design, where we tested and simulated small modules and made sure that they worked correctly before integrating them with the complete design.

We also had issues with the FPGA boards in the lab. Some of the boards do not recognize USB devices (both using the direct port and USB hub).

10 Lessons Learned

The NUNY video game is implemented as a mixed software-hardware system. Software and hardware are used for different assignments due to their different natures. We used the software for the controlling of the game logic to take advantage of its flexibility. On the other hand, the hardware was used for the display of graphics and the play of audio sounds. A wise partitioning between software and hardware is crucial for the feasibility and quality of the whole project. The interface between software and hardware should be specified as soon as possible to enable the implementations of software and hardware carry out in parallel. In the implementation of the NUNY video game, the wii controller part encountered unexpected difficulty and suffered from some delay. However, as we already specified the interface, the hardware implementation can be carried out without waiting for the software, which is the key that we can follow all the milestones we set at the beginning. Also, System Console was quite helpful for the testing of hardware without the need for the support from software.

11 Advice for Future Work

- Getting the graphics to appear clearly on the screen is a tricky process so more attention may be needed there in future projects.
- The audio implementation can be further optimized for space by using read/write buffers in the hardware and interrupts from the hardware to software.
- The initial part of getting the wiimote to connect properly can be tricky. Perhaps have a couple of people working on that initially. Once the connection is successful, the remaining code is simple.

12 Contributions

Kuangya Zhai	Wii controller, Game Logic Controller
Kshitij Bhardwaj	Linux Drivers, part of Game Logic Controller, Sprite Controller, Audio Controller
Van Bui	Image and Audio processing, part of audio controller
Vinti Vinti	Sprite Controller, part of audio controller, part of graphics preparation

Table 2: Contributions of NUNY Video Game

13 milestones

Milestone	**Date**	**Goal**	**Accomplishment**
Milestone 1	April 2	Initial integration of the audio, video and game logic modules.	The program can show moving sprites (controller through software) on the screen and play basic a beep sound.
Milestone 1	April 16	Integrate wii controller code to the existing code base. A "Hello World" version of the game.	Finalized the background and multiple sprites to be used in the final program. Achieved initial integration between software and hardware. Successfully connected the wiimote to the SoCKit board at the last minute.
Milestone 1	April 30	Implementation of the game with three levels of difficulty. Test that the game console works properly via simulation and real-time testing.	Implemented the game with difficulty by changing the number and speed of sprites and also game selection. Fully integrated the software with the hardware. Some minor bugs to be fixed.
Deadline	May 14	Finish up the project. Present and write the report.	As planned!

Table 3: Milestones of NUNY Video Game

14 References

[1] Terasic, *SoCKit User Manual*.

[2] BlueZ, “Official linux bluetooth protocol stack.” http://www.bluez.org.

[3] libwiimote, “Simple wiimote library for linux.” http://libwiimote.sourceforge.net.

15 C Code

```c
/**@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 640
#define CANVAS_SIZE_Y 480

//! so that a free dropping object shows in the screen for around 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
```

../software_cleaned/configuration.h

```
/**@file gamelogic.h
 * @brief the struct difinitions for the gamelogic and the exposed functions to operate on gamelogic
 */

#ifndef GAMELOGIC_H_
#define GAMELOGIC_H_

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

#include "configuration.h"
#include "vga_led.h"
#include "wiicontroller.h"


/**@brief encapsulate the information need for sprites
 */
typedef struct {
    bool is_on; // whether this sprite should be displayed

    double x, y; // the x, y oridinates of sprite

    double vx, vy; // the speed in x, y direction

    sprite_type my_type; // the type of the sprite

    bool is_pointed; // whether the ninja is intersect with the sprit
} sprite;


/**@brief all the information need for the game
 */
typedef struct {
    screen cur_screen;

    difficulty_level level;

    unsigned int remaining_lifes;

    unsigned int score;

    unsigned int time;

    unsigned int result;

    //! the current position of the ninja
    unsigned int ninja_x, ninja_y;

    //! the last known position of ninja. Used in case of signal losing
    unsigned int last_x, last_y;

    //! the array containing pointers to sprites
    sprite *sprites[MAX_CONCURRENT_SPRITE];
} gamelogic;


// ------- the functions operating on the game logic ----------

gamelogic *gl_init();

bool gl_update(gamelogic *pgl, wiimote_t *pwii);

void gl_start_selection(gamelogic *pgl);

void gl_end_screen(gamelogic *pgl);

void gl_reset(gamelogic *pgl);

void gl_move_ninja(gamelogic *pgl, wiimote_t *pwii);

#endif
```

../software_cleaned/gamelogic.h

```
/**@file gamelogic.c
 * @brief the implementation of the exposed functions operating on the game logic
 */

#include "gamelogic.h"

//! the multipler to be applied on the possibility and speed of sprites to control the difficulty of different levels
float MULTIPLIER = 0.0;


/**@brief initialize a new sprite and give it random initial position and speed
 */
sprite *sp_init(const sprite_type spt)
{
    sprite *psp = (sprite*)malloc(sizeof(sprite));

    psp->my_type = spt;

    psp->x = rand() % CANVAS_SIZE_X;
    psp->y = CANVAS_SIZE_Y;

    psp->vx = INIT_VX[rand() % 5];
    psp->vy = -3 * INIT_VY[rand() % 5];

    psp->vx *= MULTIPLIER; // adjust speed according to the difficulty level
    psp->vy *= MULTIPLIER;

    if(psp->x > CANVAS_SIZE_X / 2){// if the sprit comes from the right part, make it move left-ward
        psp->vx = -psp->vx;
    }

    return psp;
}


/**@brief initialize the gamelogic object
 */
gamelogic *gl_init()
{
    printf("line 9\n");
    gamelogic *pgl = (gamelogic*)malloc(sizeof(gamelogic));

    pgl->cur_screen = SELECTION;
    pgl->level = EASY;
    pgl->score = 0;
    pgl->time = 0;
    pgl->result = 0;
    pgl->remaining_lifes = 3;
    pgl->ninja_x = CANVAS_SIZE_X / 2;
    pgl->ninja_y = CANVAS_SIZE_Y / 2;

    size_t i = 0;
    for(i=0; i<MAX_CONCURRENT_SPRITE; ++i)
        pgl->sprites[i] = sp_init(i);

    return pgl;
}


/**@brief reset the value in the gamelogic module
 */
void gl_reset(gamelogic *pgl)
{
    pgl->cur_screen = SELECTION;
    pgl->level = EASY;
    pgl->score = 0;
    pgl->time = 0;
    pgl->result = 0;
    pgl->remaining_lifes = 3;
    pgl->ninja_x = CANVAS_SIZE_X / 2;
    pgl->ninja_y = CANVAS_SIZE_Y / 2;

    size_t i = 0;
    for(i=0; i<MAX_CONCURRENT_SPRITE; ++i)
        pgl->sprites[i]->is_on = false;
}


/**@brief give a sprite a new life
 */
void sp_renew(sprite *psp)
{
    psp->is_on = true;

    psp->x = rand() % CANVAS_SIZE_X;
    psp->y = CANVAS_SIZE_Y;

    psp->vx = INIT_VX[rand() % 5];
    psp->vy = -3 * INIT_VY[rand() % 5];

    psp->vx *= MULTIPLIER; // adjust speed according to the difficulty level
    psp->vy *= MULTIPLIER;

    if(psp->x > CANVAS_SIZE_X / 2){// if the sprit comes from the right part, make it move left-ward
        psp->vx = -psp->vx;
    }
```

```
}


/**@brief update the position of the ninja (stabilized)
 */
void gl_move_ninja(gamelogic *pgl, wiimote_t *pwii)
{
    unsigned int new_x, new_y;
    wii_getpos(pwii, &(new_x), &(new_y));

    bool get_new_pos = true;
    if(new_x == 9999 || new_y == 9999){
        get_new_pos = false;
    }

    if(!get_new_pos){
        new_x = pgl->last_x;
        new_y = pgl->last_y;
    }
    else{
        pgl->last_x = new_x;
        pgl->last_y = new_y;
    }

    new_x = CANVAS_SIZE_X - new_x;

    int diff_x = (int)new_x - (int)pgl->ninja_x;
    int diff_y = (int)new_y - (int)pgl->ninja_y;

    if(diff_x > MAX_DIFF)
        pgl->ninja_x += MAX_DIFF;
    else if(diff_x < -MAX_DIFF)
        pgl->ninja_x -= MAX_DIFF;
    else
        pgl->ninja_x += diff_x;

    if(diff_y > MAX_DIFF)
        pgl->ninja_y += MAX_DIFF;
    else if(diff_y < -MAX_DIFF)
        pgl->ninja_y -= MAX_DIFF;
    else
        pgl->ninja_y += diff_y;

}


/**@brief update the state of the sprite
 *
 * update the position of the sprite according to the previous speed, position
 * and gravity
 *
 */
void sp_move(sprite *psp)
{
    //psp->x = (psp->vx + psp->x) > 640 ? 0 : (psp->vx + psp->x);

    psp->x = psp->vx + psp->x;
    if(psp->x > CANVAS_SIZE_X){
        psp->x = 2 * CANVAS_SIZE_X - psp->x;
        psp->vx = -psp->vx;
    }
    else if(psp->x < 0){
        psp->x = -psp->x;
        psp->vx = -psp->vx;
    }

    psp->y = psp->vy + psp->y;
    psp->vy = psp->vy + GRAVITY * MULTIPLIER * MULTIPLIER;
}


/**@brief whether the ninja intersects with a sprite
 */
bool is_intersect(sprite *psp, gamelogic *pgl)
{
    if(!(pgl->ninja_x < 640 && pgl->ninja_x > 0))
        return false;
    if(!(pgl->ninja_y < 480 && pgl->ninja_y > 0))
        return false;

    double sqx = (psp->x - pgl->ninja_x) * (psp->x - pgl->ninja_x);
    double sqy = (psp->y - pgl->ninja_y) * (psp->y - pgl->ninja_y);

    if((sqx + sqy) < 1000){
        return true;
    }

    return false;
}


/**@brief update the state of the gamelogic. Should be called each update of the time
 *
 * update ninja and sprites positions
 * judge intersection, update game score, generates new sprites
 *
 * @return true if cutting an object, false otherwise
 */
bool gl_update(gamelogic *pgl, wiimote_t *pwii)
{
    bool sprite_intersected = false;

    gl_move_ninja(pgl, pwii);

    // update the position of all sprits
    size_t i=0;
    for(i=0; i<MAX_CONCURRENT_SPRITE; ++i){
        sprite *psp = pgl->sprites[i];

        if(psp == NULL || psp->is_on == false) continue;

        sp_move(psp); // update the position of sprite

        // whether sprite cut by ninja
        if(is_intersect(pgl->sprites[i], pgl)){
            psp->is_pointed = true;
            //play_sound
            sprite_intersected = true;
        }
        else{
            if(psp->is_pointed == true){ // current out of sprite, after cut by the ninja
                // update the score
                psp->is_pointed = false;
                pgl->score += SPRITE_SCORE[psp->my_type];

                if(psp->my_type == BOMB){// hit a bomb
                    pwii->rumble = 1;// enable the rumble
                    wiimote_update(pwii);

                    pgl->remaining_lifes--;

                    size_t i = 0;
                    for(i=0; i<MAX_CONCURRENT_SPRITE; ++i){
                        pgl->sprites[i]->is_on = false;
                    }

                    sleep(1);
                    pwii->rumble = 0;
                    wiimote_update(pwii);
                }

                psp->is_on = false; // once cut, disable the sprite
            }
        }

        // if a sprite falls below y == 0, remove from array
        if(psp != NULL && psp->y >= CANVAS_SIZE_Y+LOWER_THRESHOLD){
            psp->is_on = false;

            // the sprite is moving downward
            if(psp->vy > 0 && psp->my_type != BOMB){
                pgl->remaining_lifes--;
            }
        }
    }

    // generate new sprites according to the possibility of each sprite
    for(i=0; i<MAX_CONCURRENT_SPRITE; ++i) {

        if((pgl->sprites[i])->is_on == true) continue;

        float r = (float)rand() / RAND_MAX;

        if(r < (POSSIBILITY_MUL * MULTIPLIER * POSSIBILITY_SPRITES[i])){
            sp_renew(pgl->sprites[i]);
        }
    }

    return sprite_intersected;
}


/**@brief initialize the game logic for the screen of selection
 *
 * The positions for the options currently are not configurable
 * all magic numbers here
 */
void gl_start_selection(gamelogic *pgl)
{
    size_t i;
    for(i=0; i<3; ++i){
        pgl->sprites[i] = sp_init(i);
        pgl->sprites[i]->x = POS_SELECTIONS_X[i];
        pgl->sprites[i]->y = POS_SELECTIONS_Y[i];
    }
}


/**@brief set the first sprit to show the try-again button
 */
void gl_end_screen(gamelogic *pgl)
{
    pgl->sprites[0]->is_on = true;
    pgl->sprites[0]->x = POS_TRY_AGAIN_X;
    pgl->sprites[0]->y = POS_TRY_AGAIN_Y;
}
```

../software_cleaned/gamelogic.c

```
/**@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 *);
```

../software_cleaned/wiicontroller.h

```
/**@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"


wiimote_t wii_connect()
{
    wiimote_t wiimote = WIIMOTE_INIT;

    // the address of the wiimote is fixed here
    char *bdaddr = "2C:10:C1:8F:D0:0F";

    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(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;
}


void wii_disconnect(wiimote_t *pwiimote){
    wiimote_disconnect(pwiimote);
}
```

../software_cleaned/wiicontroller.c

```
/**@file vga_led.h
 * @brief the header for the device driver for the VGA LED Emulator
 */

#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>
#include "configuration.h"

#define VGA_LED_DIGITS 2
#define RADIUS 32

typedef struct {
    unsigned char digit;
    unsigned int segments;
} vga_led_arg_t;


#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 *)

#endif
```

../software_cleaned/vga_led.h

```
/**@file vga_leg.c
 * @brief Device driver for the VGA LED Emulator
 *
 * A Platform device implemented using the misc subsystem
 * original implemented by Stephen A. Edwards, Columbia University
 * modified by Kshitij Bhardwaj, Kuangya Zhai
 */

#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 */
  u16 segments[2 + 2*MAX_CONCURRENT_SPRITE + 2];
} 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, u16 segments)
  iowrite16(segments, dev.virtbase + digit*2);
  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;

  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 > 15)
      return -EINVAL;
            write_digit(vla.digit, vla.segments);
    break;

  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 int welcome_message[VGA_LED_DIGITS] = {
    0x003E, 0x007D};
  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,
```

```c
    .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");
```

../software_cleaned/vga_led.c

```c
/**@file audio_emulator.h
 * @brief the header for the device driver for the AUDIO Emulator
 */

#ifndef _AUDIO_H
#define _AUDIO_H

#include <linux/ioctl.h>

#define AUDIO_DIGITS 2

typedef struct {
  unsigned char digit;
  unsigned int segments;
} audio_arg_t;

#define AUDIO_MAGIC 'q'

/* ioctls and their arguments */
#define AUDIO_WRITE_DIGIT _IOW(AUDIO_MAGIC, 1, audio_arg_t *)
#define AUDIO_READ_DIGIT  _IOWR(AUDIO_MAGIC, 2, audio_arg_t *)

#endif
```

../software_cleaned/audio_emulator.h

```c
/**@file audio_emulator.c
 * @brief Device driver for the AUDIO Emulator
 *
 * A Platform device implemented using the misc subsystem
 * Devired from the vga_led.c file originally developed by Stephen A. Edwards, Columbia University
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
```

```c
#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 "audio_emulator.h"

#define DRIVER_NAME "audio_emulator"

/*
 * Information about our device
 */
struct audio_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers can be accessed in memory */
    u16 segments[2];
} 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, u16 segments)
{
    iowrite16(segments, dev.virtbase + digit*2);
    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 audio_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    audio_arg_t vla;

    switch (cmd) {
    case AUDIO_WRITE_DIGIT:
        if (copy_from_user(&vla, (audio_arg_t *) arg,
                   sizeof(audio_arg_t)))
            return -EACCES;
        if (vla.digit > (2))
            return -EINVAL;
        write_digit(vla.digit, vla.segments);
        break;
    default:
        return -EINVAL;
    }

    return 0;
}

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

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

```
/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init audio_probe(struct platform_device *pdev)
{
    /*static unsigned int welcome_message[VGA_LED_DIGITS] = {
        0x003E, 0x007D};*/
    int i, ret;

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

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

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

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

    return 0;

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

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

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

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver audio_driver = {
    .driver = {
        .name    = DRIVER_NAME,
        .owner   = THIS_MODULE,
        .of_match_table = of_match_ptr(audio_of_match),
```

```c
    },
    .remove = __exit_p(audio_remove),
};

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

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

module_init(audio_init);
module_exit(audio_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("Audio Emulator");
```

../software_cleaned/audio_emulator.c

```c
/**@file main.c
 * @brief this file contains the main() function
 */

#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
#include <poll.h>
#include <signal.h>
#include <assert.h>

#include "vga_led.h"
#include "audio_emulator.h"

#include "configuration.h"
#include "wiicontroller.h"
#include "gamelogic.h"

#define BUFFER_SIZE     32768       // 32 KB buffers

/** Buffer format specifier. */
#define AL_FORMAT_MONO8                          0x1100
#define AL_FORMAT_MONO16                         0x1101
#define AL_FORMAT_STEREO8                        0x1102
#define AL_FORMAT_STEREO16                       0x1103

int vga_led_fd;
int audio_fd;


void write_segment_vga(gamelogic *pgl)
{
    vga_led_arg_t vla2;

    int i;
    int j = 0;
```

```
//--------------- writing the position of ninja ---------------
vla2.digit = j++;
vla2.segments = pgl->ninja_x;
if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla2)) {
    perror("ioctl(VGA_LED_WRITE_DIGIT) failed ninjaX");
    return;
}
vla2.digit = j++;
vla2.segments = pgl->ninja_y;
if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla2)) {
    perror("ioctl(VGA_LED_WRITE_DIGIT) failed ninjaY");
    return;
}

//--------------- writing the position of sprites ---------------
for (i = 0; i < (MAX_CONCURRENT_SPRITE); i++){
    int x_tmp = 999, y_tmp = 999;
    if(pgl->sprites[i] != NULL && pgl->sprites[i]->is_on){
        x_tmp = (int)((pgl->sprites[i])->x);
        y_tmp = (int)((pgl->sprites[i])->y);
    }

    vla2.digit = j++;
    vla2.segments = x_tmp;
    if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla2)) {
        perror("ioctl(VGA_LED_WRITE_DIGIT) failed spriteX");
        exit(1);
        return;
    }

    vla2.digit = j++;
    vla2.segments = y_tmp;
    if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla2)) {
        perror("ioctl(VGA_LED_WRITE_DIGIT) failed spriteY");
        exit(1);
        return;
    }
}

assert(j == 12);
//------------ digit[12] : current screen, win/fail game result -------------
vla2.digit = j++;
unsigned int b_tmp = 0x0000; // 16 bit uint
switch (pgl->cur_screen) {
    case SELECTION:
        b_tmp = 0x0000;
        break;
    case PLAY:
        switch (pgl->level) {
            case 0:
                b_tmp = 0x0005;
                break;
            case 1:
                b_tmp = 0x0009;
                break;
            case 2:
                b_tmp = 0x0011;
                break;
            default:
                b_tmp = 0x0005;
                break;
        }
        break;
    case RESULT:
        if (pgl->result == 0)
            b_tmp = 0x0002;
        else{
                if (pgl->level == 2)
                    b_tmp = 0x0032;
                else
                    b_tmp = 0x002E;
            }

            break;
        default:
            b_tmp = 0x0000;
            break;
    }
    vla2.segments = b_tmp;
    if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla2)) {
        perror("ioctl(VGA_LED_WRITE_DIGIT) failed control segment");
        exit(1);
        return;
    }

    //------------- digit[13] : score ---------------
    vla2.digit = j++;
    vla2.segments = pgl->score;
    if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla2)) {
        perror("ioctl(VGA_LED_WRITE_DIGIT) failed score");
        exit(1);
        return;
    }

    //----------- digit[14] : remaining life ------------
    vla2.digit = j++;
    switch (pgl->remaining_lifes){
        case 0: b_tmp = 0x0007;
                break;
        case 1: b_tmp = 0x0006;
                break;
        case 2: b_tmp = 0x0004;
                break;
        case 3: b_tmp = 0x0000;
                break;
        default: b_tmp = 0x0000;
                 break;
    }
    vla2.segments = b_tmp;
    if (ioctl(vga_led_fd, VGA_LED_WRITE_DIGIT, &vla2)) {
        perror("ioctl(VGA_LED_WRITE_DIGIT) failed score");
        exit(1);
        return;
    }
}


void write_segment_vga_audio(const unsigned int segs[2])
{
    audio_arg_t vla;
    int i;
    for (i = 0 ; i < 2; i++) {
        vla.digit = i;
        vla.segments = segs[i];
        if (ioctl(audio_fd, AUDIO_WRITE_DIGIT, &vla)) {
            perror("ioctl(AUDIO_WRITE_DIGIT) failed");
            return;
        }
    }
}


int get_audio_data(unsigned int *audio_data)
{
  char *audio_data_file = "gong.txt";
  FILE *fp;
  char *mode = "r";
  int packets_audio_file = 0;

  unsigned int temp_audio;

  int rv = 1;
        fp = fopen(audio_data_file, mode);
        if (fp == NULL){
            printf("ERROR opening file\n");
            exit(1);
        }

  while (rv != EOF){
    rv = fscanf(fp, "%x", &temp_audio);
    if (rv != EOF){
      packets_audio_file++;
    }

  }
  fclose(fp);


  unsigned int audio_array[packets_audio_file];

  rv = 1;
        fp = fopen(audio_data_file, mode);
        if (fp == NULL){
            printf("ERROR opening file\n");
            exit(1);
        }
  int c = 0;
  while (rv != EOF){
    rv = fscanf(fp, "%x", &temp_audio);
    if (rv != EOF){
      audio_array[c] = temp_audio;
      c++;
    }

  }
  fclose(fp);

  audio_data = audio_array;
  return packets_audio_file;
}


/**@brief the game controller entry point
 *
 * This function contains the main routine of the NUNY video game
 *
 * @return 0 for normal execution, other values for error codes
 */
int main()
{
    wiimote_t wiimote = wii_connect();
    gamelogic *pgl = gl_init();

    srand(time(NULL));

    bool is_sprite_intersect = false;

    unsigned int *audio_data;
    int packets_audio_file = get_audio_data(&audio_data);

    vga_led_arg_t vla;
    audio_arg_t ala;
    unsigned int adcur[2];
    static const char filename[] = "/dev/vga_led";
    static const char filename2[] = "/dev/audio_emulator";

    printf("VGA LED Userspace program started\n");
    if ( (vga_led_fd = open(filename, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }
    if ( (audio_fd = open(filename2, O_RDWR)) == -1) {
        fprintf(stderr, "could not open %s\n", filename);
        return -1;
    }

    // ---------------------------------
    //           Selection Screen
    // ---------------------------------
TAG_SEL:
    pgl->cur_screen = SELECTION;
    gl_start_selection(pgl);
    write_segment_vga(pgl);

    while (wiimote_is_open(&wiimote)){

        gl_move_ninja(pgl, &wiimote);

        write_segment_vga(pgl);

        size_t i;
        for(i=0; i<LEVELS; ++i){
            if(is_intersect(pgl->sprites[i], pgl)){
                pgl->level = i;
                MULTIPLIER = MULTIPLIERS[i];
                break;
            }
        }

        if(i != LEVELS) // goto next screen
            break;
    }

    printf("Level Selected: %u\n", pgl->level);

    write_segment_vga(pgl);

    // ---------------------------------
    //           Game Screen
    // ---------------------------------
TAG_PLAY:
    gl_reset(pgl);

    float cur_utime = 0.0f;
    pgl->cur_screen = PLAY;
    printf("game: Current screen: %u\n", pgl->cur_screen);

    //Keep audio off by default
    adcur[0] = 0;
    write_segment_vga_audio(adcur);

    printf("game: Current Level: %u\n", pgl->level);
    while (wiimote_is_open(&wiimote) && pgl->time < GAMETIME){

        // update the gamelogic state
        is_sprite_intersect = gl_update(pgl, &wiimote);

        if(!is_sprite_intersect){
            adcur[0] = 0;
            write_segment_vga_audio(adcur);
        }
        else{
            adcur[0] = 1;
            write_segment_vga_audio(adcur);
        }

        write_segment_vga(pgl);

        if (pgl->score >= TARGET || (pgl->remaining_lifes == 0)){

            goto TAG_RES;

            if (pgl->remaining_lifes == 0)
                pgl->result = 0;
            else
                pgl->result = 1;

            pgl->cur_screen = RESULT;
            write_segment_vga(pgl);

            gl_end_screen(pgl);

            while(wiimote_is_open(&wiimote)){

                gl_move_ninja(pgl, &wiimote);
                write_segment_vga(pgl);

                if(is_intersect(pgl->sprites[0], pgl)){
                    goto TAG_SEL;
                }

            }

        }

        // ---- update time time counter ---- //
        cur_utime += 1;
        if(cur_utime >= 100){// 1 second passed, update the displayed time
            pgl->time++;
            cur_utime = 0;
        }
    }

    // ----------------------------------
    //           Result Screen
    // ----------------------------------
TAG_RES:
    if (pgl->remaining_lifes == 0)
        pgl->result = 0;
    else
        pgl->result = 1;

    pgl->cur_screen = RESULT;
    write_segment_vga(pgl);

    gl_end_screen(pgl);

    while(wiimote_is_open(&wiimote)){

        gl_move_ninja(pgl, &wiimote);
        write_segment_vga(pgl);

        if(is_intersect(pgl->sprites[0], pgl)){
            goto TAG_SEL;
        }

    }
    wii_disconnect(&wiimote);
    printf("VGA LED Userspace program terminating\n");
    return 0;
}
```

../software_cleaned/main.c

```
ifneq (${KERNELRELEASE},)

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

else

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

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


CC:=gcc
DEFS:=-D_ENABLE_TILT -D_ENABLE_FORCE
CFLAGS:=-Wall -pipe $(DEFS) -g -O9 -Os
INCLUDES:=-I./src
LIBS:=-L./lib -lcwiimote -lbluetooth -lm

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

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

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

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

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

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

endif
```

../software_cleaned/Makefile

```
/**@file readogg.c
 * @brief Decodes Ogg files
 *
 * Decodes Ogg files using Ogg Vorbis SDK. Partially converts to MIF format.
 * Original code written by Anthony Yuen,
 * http://archive.gamedev.net/archive/reference/articles/article2031.html
 * Modified by Van Bui
 */

#include <stdio.h>
#include <string.h>

#include <vorbis/vorbisfile.h>

#define BUFFER_SIZE     32768       // 32 KB buffers
```

```
/** Buffer format specifier. */
#define AL_FORMAT_MONO8                          0x1100
#define AL_FORMAT_MONO16                         0x1101
#define AL_FORMAT_STEREO8                        0x1102
#define AL_FORMAT_STEREO16                       0x1103

#define BIG_ENDIAN

int LoadOGG(char *fileName, char *buffer, int *format, int *freq)
{
  int endian = 1;                // 0 for Little-Endian, 1 for Big-Endian
  int bitStream;
  long bytes;
  char array[BUFFER_SIZE];        // Local fixed size array
  FILE *f;
  int i;
  int offset;
  int numbytes;

  numbytes=0;
  offset=0;

  // Open for binary reading
  f = fopen(fileName, "rb");

  if (f == NULL)
   {
     printf( "Cannot open file!\n");
     exit(-1);
   }
  // end if

  vorbis_info *pInfo;
  OggVorbis_File oggFile;

  // Try opening the given file
  if (ov_open(f, &oggFile, NULL, 0) != 0)
    {
      printf( "Error opening file for decoding...");
      exit(-1);
    }
  // end if

 // Get some information about the OGG file
  pInfo = ov_info(&oggFile, -1);

  // Check the number of channels... always use 16-bit samples
  if (pInfo->channels == 1)
    *format = AL_FORMAT_MONO16;
  else
    *format = AL_FORMAT_STEREO16;
  // end if

  // The frequency of the sampling rate
  *freq = pInfo->rate;

  // Keep reading until all is read
  do
    {
      // Read up to a buffer's worth of decoded sound data
      bytes = ov_read(&oggFile, array, BUFFER_SIZE, endian, 2, 1, &bitStream);

      if (bytes < 0)
  {
            ov_clear(&oggFile);
            printf("Error decoding file...\n");
            exit(-1);
 }
   // end if

   // Append to end of buffer
   for (i=0; i < bytes; i++)
 buffer[i+offset]=array[i];

   numbytes=numbytes+bytes;
   offset = offset+bytes;

  }

 while (bytes > 0);

 // Clean up!
 ov_clear(&oggFile);

 return numbytes;
}


int main(int argc, char** argv)
{

  int i,j;
  int format;                          // The sound data format
  int freq;                            // The frequency of the sound data
  char bufferData[BUFFER_SIZE*100];    // The sound buffer data from file
  int numbytes;

  numbytes = LoadOGG("bomb.ogg", bufferData, &format, &freq);

  j=0;

  for (i=0; i < numbytes/2; i+=4) {
   printf("%4d :%13u;\n", j, (bufferData[i] << 8) + bufferData[i+2]);
   j++;
  }

  return 0;
}
```

../software_cleaned/readogg.c

```
% mifflegen.m
% Converts image files to MIF and also resizes image files
% Modified by: Vinti Vinti

function [outfname, rows, cols] = miffilegen(infile, outfname, numrows, numcols)

img = imread(infile);

imgresized = imresize(img, [numrows numcols]);

[rows, cols, rgb] = size(imgresized);

imgscaled = imgresized/16 -1;
imshow(imgscaled*16);

fid = fopen(outfname,'w');

fprintf(fid,'-- %3ux%3u 12bit image color values\n\n',rows,cols);
fprintf(fid,'WIDTH = 12;\n');
fprintf(fid,'DEPTH = %4u;\n\n',rows*cols);
fprintf(fid,'ADDRESS_RADIX = UNS;\n');

fprintf(fid,'DATA_RADIX = UNS;\n\n');
fprintf(fid,'CONTENT BEGIN\n');

count = 0;
for r = 1:rows
    for c = 1:cols
        red = uint16(imgscaled(r,c,1));
        green = uint16(imgscaled(r,c,2));
        blue = uint16(imgscaled(r,c,3));
        color = red*(256) + green*16 + blue;
        image2(r,c)=color;
        fprintf(fid,'%4u : %4u;\n',count, color);
        count = count + 1;
    end
end

fprintf(fid,'END;');

fclose(fid);
```

../software_cleaned/miffilegen.m

16 VHDL Code

```

//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
   input [15:0] audio_sample, //get audio sample from audio codec interface, not needed here
    output [15:0] audio_output, //sends audio sample to audio codec
    input [15:0] M_bell,    //bell sound ROM data
    input [15:0] M_city,    //city sound ROM data
    input [15:0] M_who, //whoosh sound ROM data
    input [15:0] M_sw,  //sword sound ROM data
    output [14:0] addr_bell,    //ROM addresses
    output [14:0] addr_city,
    output [14:0] addr_who,
    output [14:0] addr_sw,
    input  [3:0] control    //Control from avalon bus
);


reg  [15:0]  index = 15'd0;     //index through the sound ROM data for different sounds
reg  [15:0]  index_who = 15'd0;
reg  [15:0]  index_bell = 15'd0;
reg  [15:0]  index_sw = 15'd0;
reg [15:0] count = 15'd0;


reg [15:0] dat;

assign audio_output = dat;

//assign index to ROM addresses
always @(posedge clk) begin
```

```

    addr_bell <= index_bell;
    addr_city <= index;
    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 == 1 || count >= 1) begin
            if (index_sw <= 16537)  //play sword sound
                dat <= M_sw;
        if (index_sw == 15'd16537) begin
                index_sw <= 15'd0;
                count <= 15'd0;
            end
        else begin
            index_sw <= index_sw +1'b1; //increment sword index
                count <= count + 1'b1;
            end
        end
        if (control == 0 && count == 0) begin //play city sound
            index_sw <= 15'b0;
            dat <= M_city;
        end

    if (index == 15'd22049)
            index <= 15'd0;
    else
        index <= index +1'b1;   //increment city index

  end
  else
            dat <= 16'd0;
end

endmodule
```

../hardware_cleaned/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_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 logic    resetn,
   input logic [15:0]  writedata, //data from SW
   input logic address,   //1-bit peripheral address
   input logic write,
   input logic chipselect,
   output logic irq,     // interrupt from fpga to hps
    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 = !KEY[0];
wire main_clk;
wire audio_clk;
wire ctrl;
//wire chipselect = 1;
wire [1:0] sample_end;
wire [1:0] sample_req;
wire [15:0] audio_output;
wire [15:0] audio_sample;
wire [15:0] audio_sw;
wire [15:0] audio_ip;

//Sound samples from audio ROM blocks
wire [15:0] M_bell;
wire [15:0] M_city;
wire [15:0] M_who;
wire [15:0] M_sw;

//Audio ROM block addresses
wire [14:0] addr_bell;
wire [14:0] addr_city;
wire [14:0] addr_who;
wire [14:0] addr_sw;

//Store sounds in memory ROM blocks
bell b0 (.clock(OSC_50_B8A), .address(addr_bell), .q(M_bell));
city c0 (.clock(OSC_50_B8A), .address(addr_city), .q(M_city));
whoosh_new w0 (.clock(OSC_50_B8A), .address(addr_who), .q(M_who));
sword s0 (.clock(OSC_50_B8A), .address(addr_sw), .q(M_sw));

//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 (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’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),
    .audio_sample  (audio_sample),
    .addr_bell(addr_bell),
    .addr_city(addr_city),
    .addr_who(addr_who),
    .addr_sw(addr_sw),
    .M_bell(M_bell),
    .M_who(M_who),
    .M_city(M_city),
    .M_sw(M_sw),
   .control(ctrl)
);

//Read control (on/off) for striking sound from SW. Also has provision
//for reading audio samples from SW but not used..
 always_ff @(posedge OSC_50_B8A)
     if (resetn) begin
  ctrl <= 0;


     end
  else if (chipselect && write)
  begin

   case(address)
  1'b0: ctrl <= writedata[0]; // to turn the audio codec on/ off
  1'b1: audio_sw <= writedata;  // read audio sample (16 bits) from the software/ audio file
  endcase
  end
endmodule
```

../hardware_cleaned/Audio_Top.sv

```
/*
RGB_controller.sv
Contains the line-buffer based sprite controller, accessing various sprites and
    assigning priorities to them*/

module RGB_controller(clk,clk50,screen,
              x, y, x1,y1, x2,y2, x3,y3, x4,y4, x5,y5,
              hcount,vcount, nin_life,level,result,
              addr, addr_bg, addr_b1, addr_b2, addr_b3, addr_b4, addr_b5, addr_s, addr_sc,
    one, ten, hun,
              addr_nl, addr_t, /*addr_sun, addr_mn, addr_rn, addr_try, addr_sym*/, addr_sym,
     addr_nun, addr_b6, M_bg1, M_bg2, M_bg3, M_bg4,
              M_n1,M_n2,M_n3, M_nl1, M_nl2, M_nl3, M_sun, M_mn, M_rn, M_try, M_nun, M_sym,
    M_b6,
              M_b1,M_b2,M_b3,M_b4,M_b5, M_s1,M_s2,M_s3, M_ps, M_fl, M_dp,
                            M_sc0, M_sc1, M_sc2, M_sc3, M_sc4, M_sc5, M_sc6, M_sc7, M_sc8,
    M_sc9,
              //line_buffer,
              VGA_R, VGA_G, VGA_B
               );
```

```
 input wire [3:0] one;  //ones place of score
 input wire [3:0] ten;  //tens place of score
 input wire [3:0] hun;  //hundreds place of score
 input wire [3:0] nin_life; //remaining ninja lives
 input wire [2:0] screen;   //Which screen?
 input wire [2:0] level;    //Which level?
 inout wire result; //pass or fail result
 input  wire [10:0] hcount; //Horizontal count
 input wire [9:0]  vcount;  //Vertical count
 input wire clk,clk50;  //Main VGA clock
 input wire [11:0] M_n1; // ROM data for  3 ninja sword positions
 input wire [11:0] M_n2;
 input wire [11:0] M_n3;
 input wire [11:0] M_b1;    // ROM data for 6 moving sprites
 input wire [11:0] M_b2;
 input wire [11:0] M_b3;
 input wire [11:0] M_b4;
 input wire [11:0] M_b5;
 input wire [11:0] M_b6;
 input wire [11:0] M_bg1;   //ROM data for 4 background splits sprites
 input wire [11:0] M_bg2;
 input wire [11:0] M_bg3;
 input wire [11:0] M_bg4;
 input wire [11:0] M_s1;    //ROM data for 3 levels sprites
 input wire [11:0] M_s2;
 input wire [11:0] M_s3;
 input wire [11:0] M_ps;    //ROM data for pass sprite
 input wire [11:0] M_fl;    //ROM data for fail sprites
 input wire [11:0] M_dp;    //ROM data for diploma sprite
 input wire [11:0] M_sc0;   //ROM data for zero-nine sprite
 input wire [11:0] M_sc1;
 input wire [11:0] M_sc2;
 input wire [11:0] M_sc3;
 input wire [11:0] M_sc4;
 input wire [11:0] M_sc5;
 input wire [11:0] M_sc6;
 input wire [11:0] M_sc7;
 input wire [11:0] M_sc8;
 input wire [11:0] M_sc9;
 input wire [11:0] M_nl1;   //ROM data for 3 sprite lives
 input wire [11:0] M_nl2;
 input wire [11:0] M_nl3;
 input wire [11:0] M_sun;   //ROM data for sun sprite
 input wire [11:0] M_mn;    //ROM data for moon sprite
 input wire [11:0] M_rn;    //ROM data for rain sprite
 input wire [11:0] M_nun;   //ROM data for NUNY name sprite
 input wire [11:0] M_try;   //ROM data for try again sprite
 input wire [11:0] M_sym;   //ROM data for NUNY symbol sprite
 input wire [15:0] x,y,x1,y1,x2,y2,x3,y3,x4,y4,x5,y5;   //X,Y coordinates read from SW for
    moving sprites

 output wire [11:0] addr;  //ROM address of ninja
 output wire [11:0] addr_b1;    //ROM address of 6 moving sprites
 output wire [11:0] addr_b2;
 output wire [11:0] addr_b3;
 output wire [11:0] addr_b4;
 output wire [11:0] addr_b5;
 output wire [11:0] addr_b6;
 output wire [11:0] addr_s; //ROM address of levels sprites
 output wire [11:0] addr_sc;    //ROM address for score numbers sprites
 output wire [11:0] addr_nl;    //ROM address for ninja lives
 output wire [14:0] addr_bg;    //ROM address for background
 output wire [14:0] addr_nun;   //ROM address for NUNY name
 output wire [11:0] addr_sym;   //ROM address for NUNY symbol
 output wire [11:0] addr_t; //ROM address for sun/moon/rain
 output wire [7:0] VGA_R, VGA_G, VGA_B; //RGB output

//----------------MISC declarations-----------------------------
```

```
// local ROM addresses for various sprites:
 wire [11:0] addr_s1,addr_s2,addr_s3, addr_ps, addr_fl, addr_dp;
 wire [11:0] addr_nl1,addr_nl2,addr_nl3;
 wire [11:0] addr_suntemp;
 wire [11:0] addr_mntemp;
 wire [11:0] addr_rntemp;
 wire [11:0] addr_trytemp;
 wire [14:0] addr_nuntemp;
 wire [14:0] addr_symtemp;
 wire [11:0] addr_sc0, addr_sc1, addr_sc2, addr_sc3, addr_sc4, addr_sc5, addr_sc6, addr_sc7, addr_sc8, addr_sc9;


  reg [11:0] line_buffer [639:0];  // line buffer for sprites
  wire [10:0] xrow;     //one row in X direction of VGA display
  logic [11:0] M_bg,addr_life;  //local backgorund sprite data
  reg [11:0] M,M_l,M_b,M_s,M_pf, M_sc, M_nl, M_temp; //local sprite ROM data

  wire [11:0] M_buf; //data to be stored in buffers
  logic [9:0] yrow; //one row in Y direction of VGA display

  reg [11:0] buffer1 [639:0];   //2 buffers used to store sprites
  reg [11:0] buffer2 [639:0];

  assign xrow = (hcount >> 1);
  assign yrow = vcount;

  reg [3:0] cnt = 4'd0;     //buffer count
  reg buf_cnt;

  reg [2:0] nin_life_temp = 3'b111;

  reg [3:0] temp, temp2, temp3; //temp registers to store ones/tens/hundreds
  reg temp_fl = 0; //temp flag to store the ones/tens/hundreds flag
  reg [10:0] tempx; //temp declarations to store the x region of ones/tens/hundreds
  reg [10:0] tempy; //temp declarations to store the y region of ones/tens/hundreds
//-----------------------------------------------------------------------


  //--------Sprite region declarations--------------------------
  logic [10:0] regionx, regiony;    //ninja region
  logic [10:0] regionx1, regiony1;  //6 moving sprite regions
  logic [10:0] regionx2, regiony2;
  logic [10:0] regionx3, regiony3;
  logic [10:0] regionx4, regiony4;
  logic [10:0] regionx5, regiony5;
  logic [10:0] regionx6, regiony6;
  logic [10:0] stagex1, stagey1;    // 3 stages (or levels) regions
  logic [10:0] stagex2, stagey2;
  logic [10:0] stagex3, stagey3;
  logic [10:0] passx, passy;    //pass region
  logic [10:0] failx, faily;    //fail region
  logic [10:0] onex, oney;  //one's place of score region
  logic [10:0] tenx, teny;  //ten's place of score region
  logic [10:0] hunx, huny;  //hundred's place of score region
  logic [10:0] nin1x, nin1y;    //ninja life 1 region
  logic [10:0] nin2x, nin2y;    //ninja life 2 region
  logic [10:0] nin3x, nin3y;    //ninja life 2 region
  logic [10:0] sunx, suny;  //sun region
  logic [10:0] moonx, moony;    //moon region
  logic [10:0] rainx, rainy;    //rain region
  logic [10:0] tryx, tryy;  //try again region
  logic [10:0] nunx, nuny;  //nuny name sprite region
  logic [10:0] dipx, dipy;  //diploma region
  logic [10:0] symx, symy;  //NUNY symbol region
  //--------------------------------------------------------------------
```

```
//--------Assign sprite region base locations---------------------
assign regionx=(xrow-x);
assign regiony = (yrow-y);
assign regionx1=(xrow-x1);
assign regiony1 = (yrow- y1);
assign regionx2=(xrow-x2);
assign regiony2 = (yrow- y2);
assign regionx3=(xrow-x3);
assign regiony3 = (yrow- y3);
assign regionx4=(xrow-x4);
assign regiony4 = (yrow- y4);
assign regionx5=(xrow-x5);
assign regiony5 = (yrow- y5);
assign regionx6=(xrow-x5);
assign regiony6 = (yrow- y5);
assign stagex1=(xrow-11'd187);
assign stagey1 = (yrow- 10'd300);
assign stagex2=(xrow-11'd287);
assign stagey2 = (yrow- 10'd300);
assign stagex3=(xrow-11'd387);
assign stagey3 = (yrow- 10'd300);
assign passx=(xrow-11'd187);
assign passy = (yrow- 10'd150);
assign failx=(xrow-11'd287);
assign faily = (yrow- 10'd150);
assign onex=(xrow-11'd90);
assign oney = (yrow);
assign tenx=(xrow-11'd50);
assign teny = (yrow);
assign hunx=(xrow-11'd10);
assign huny = (yrow);
assign nin1x=(xrow-11'd480);
assign nin1y = (yrow);
assign nin2x=(xrow-11'd520);
assign nin2y = (yrow);
assign nin3x=(xrow-11'd560);
assign nin3y = (yrow);
assign sunx=(xrow-11'd483);
assign suny = (yrow-11'd50);
assign moonx=(xrow-11'd481);
assign moony = (yrow-11'd50);
assign rainx=(xrow-11'd481);
assign rainy = (yrow-11'd50);
assign tryx=(xrow-11'd481);
assign tryy = (yrow-11'd50);
assign nunx=(xrow-11'd203);
assign nuny = (yrow-11'd50);
assign dipx=(xrow-187);
assign dipy = (yrow-150);
assign symx=(xrow-11'd135);
assign symy = (yrow-11'd50);
//-----------------------------------------------------------

//-------------sprite on flags---------------------------
logic ninja;
logic sky;
logic black;
logic skyline;
logic book;
logic book1;
logic book2;
logic book3;
logic book4;
logic book5;
logic book6;
logic dip_fl;
logic bg1,bg2,bg3,bg4;
wire life;
```

```
  wire stage1,stage2,stage3;
  wire pass_fl, fail_fl;
  wire one_fl, ten_fl, hun_fl;
  wire nin1_fl, nin2_fl, nin3_fl;
  wire sun_fl, moon_fl, rain_fl;
  wire nun_fl, try_fl, sym_fl;
 //----------------------------------------------------------


//-----------------Sprite flags switched ON if inside sprite region-----------------------------------
  assign sky = (yrow <= 154)?1'b1:1'b0;
  assign skyline = ((yrow>= 155 )&&(yrow <= 353))?1'b1:1'b0;
  assign black = ((yrow>= 354 ))?1'b1:1'b0;
  assign book1 = (screen[0] && regionx1[10:6]==0 && regiony1[10:6]==0)?1'b1:1'b0;
  assign book2 = (screen[0] && regionx2[10:6]==0 && regiony2[10:6]==0)?1'b1:1'b0;
  assign book3 = (screen[0] && regionx3[10:6]==0 && regiony3[10:6]==0)?1'b1:1'b0;
  assign book4 = (screen[0] && regionx4[10:6]==0 && regiony4[10:6]==0)?1'b1:1'b0;
  assign book5 = (screen[0] && regionx5[10:6]==0 && regiony5[10:6]==0 && (level[1] == 1 || level[0] == 1))?1'b1:1'b0;
  assign book6 = (screen[0] && regionx6[10:6]==0 && regiony6[10:6]==0 && (level[2] == 1))?1'b1:1'b0;
  assign dip_fl = (screen[2] && dipx[10:6]==0 && dipy[10:6]==0 && level[2] == 1 && result == 1)?1'b1:1'b0;
  assign stage1 = (screen[1] && stagex1[10:6]==0 && stagey1[10:6]==0)?1'b1:1'b0;
  assign stage2 = (screen[1] && stagex2[10:6]==0 && stagey2[10:6]==0)?1'b1:1'b0;
  assign stage3 = (screen[1] && stagex3[10:6]==0 && stagey3[10:6]==0)?1'b1:1'b0;
  assign nun_fl = (screen[1] && (xrow >= 205 && xrow <= 403) && (yrow >= 50 && yrow <= 95))?1'b1:1'b0;
  assign try_fl = (screen[2] && tryx[10:6]==0 && tryy[10:6]==0)?1'b1:1'b0;
  assign pass_fl = (screen[2] && passx[10:6]==0 && passy[10:6]==0 && result==1 && (level[1] == 1 || level[0] == 1))?1'b1:1'b0;
  assign fail_fl = (screen[2] && failx[10:6]==0 && faily[10:6]==0 && result==0)?1'b1:1'b0;
  assign ninja = (regionx[10:6]==0 && regiony[10:6]==0)?1'b1:1'b0;
  assign one_fl = (onex[10:5]==0 && oney[10:5]==0)?1'b1:1'b0;
  assign ten_fl = (tenx[10:5]==0 && teny[10:5]==0)?1'b1:1'b0;
  assign hun_fl = (hunx[10:5]==0 && huny[10:5]==0)?1'b1:1'b0;
  assign nin1_fl = (nin1x[10:5]==0 && nin1y[10:5]==0 && (nin_life[0] == 0))?1'b1:1'b0;
  assign nin2_fl = (nin2x[10:5]==0 && nin2y[10:5]==0 && nin_life[1] == 0)?1'b1:1'b0;
  assign nin3_fl = (nin3x[10:5]==0 && nin3y[10:5]==0 && nin_life[2] == 0)?1'b1:1'b0;
  //assign sun_fl = (screen[0] && sunx[10:6]==0 && suny[10:6]==0 && level[2] == 1)?1'b1:1'b0;
  assign sun_fl = (screen[0] && (xrow >= 483 && xrow <= 547) && (yrow >= 50 && yrow <= 114) && level[2] == 1)?1'b1:1'b0;
  assign sym_fl = (symx[10:6]==0 && symy[10:6]==0 && screen[1])?1'b1:1'b0;
  assign moon_fl = (moonx[10:6]==0 && moony[10:6]==0 && level[1] == 1 && screen[0])?1'b1:1'b0;
  assign rain_fl = (rainx[10:6]==0 && rainy[10:6]==0 && level[0] == 1 && screen[0])?1'b1:1'b0;
  assign book = (book1 || book2 || book3 || book4 || book5 || book6);
//---------------------------------------------------------------------


//--------Reading sprite ROM data into a local reg when sprite flag is ON--------

//sun/moon/rain/tryagain sprites
    always @(*) begin
        if (try_fl)
            M_temp = M_try;
        else if (moon_fl)
            M_temp = M_mn;
        else if (sun_fl)
            M_temp = M_sun;
        else if (rain_fl)
            M_temp = M_rn;
    end

//ninja lives
```

```
    always @(*) begin
        if (nin1_fl)
            M_nl = M_nl1;
        else if (nin2_fl)
            M_nl = M_nl2;
        else if (nin3_fl)
            M_nl = M_nl3;
    end

//number ones/tens/hundreds sprites
    always @(*) begin
        if (one_fl)
            temp = one;
        else if (ten_fl)
            temp = ten;
        else if (hun_fl)
            temp = hun;
        else
            temp = one;
        case (temp)
            4'd0: M_sc = M_sc0;
            4'd1: M_sc = M_sc1;
            4'd2: M_sc = M_sc2;
            4'd3: M_sc = M_sc3;
            4'd4: M_sc = M_sc4;
            4'd5: M_sc = M_sc5;
            4'd6: M_sc = M_sc6;
            4'd7: M_sc = M_sc7;
            4'd8: M_sc = M_sc8;
            4'd9: M_sc = M_sc9;
            default: M_sc = M_sc0;
        endcase
    end

    // selecting background sprites
  always @(*)
    begin
      if (skyline==1) begin
      //background sprite 5
        if ((xrow>= 0 )&& (xrow <= 159)) begin
          M_bg = M_bg1;
         end
      //backgroung sprite 3
      else if ((xrow>= 160 )&& (xrow <= 320)) begin
         M_bg = M_bg2;
        end
     if ((xrow>= 321 )&& (xrow <= 480)) begin
          M_bg = M_bg3;
         end
      //backgroung sprite 4
      else if ((xrow>= 481 )&& (xrow <= 639)) begin
         M_bg = M_bg4;
        end
        end
        //M_bg = 12'd0;
    end

  //selection of moving sprites
  always @(*) begin
    if ((book1==1) && M_b1!=12'd4095) begin
      M_b = M_b1;
    end
    else if ((book2==1) && M_b2!=12'd4095) begin
      M_b = M_b2;
    end
    else if ((book3==1) && M_b3!=12'd4095) begin
      M_b = M_b3;
    end
    else if ((book4==1) && M_b4!=12’d3567) begin
      M_b = M_b4;
    end
    else if ((book5==1) && M_b5!=12’d0000) begin
      M_b = M_b5;
        end
    else if ((book6==1) && M_b6!=12’d0000) begin
      M_b = M_b6;
    end else
      M_b = 12’d4095;
    end

  //selecting stage selection/pass/fail/diploma sprites
  always @(*) begin
    if ((stage1==1)) begin
      M_s = M_s1;
    end
    else if ((stage2==1)) begin
      M_s = M_s2;
    end
    else if ((stage3==1)) begin
      M_s = M_s3;
    end
    else if ((pass_fl==1)) begin
      M_s = M_ps;
    end
    else if ((fail_fl==1)) begin
      M_s = M_fl;
      end
    else if ((dip_fl==1)) begin
      M_s = M_dp;
    end
    end


  // ninja sprite selection of sword position
  always_ff @(posedge clk)
  begin
   case(cnt)
  4’d0: M <= M_n1;
  4’d1: M <= M_n1;
  4’d2: M <= M_n1;
  4’d3: M <= M_n1;
  4’d4: M <= M_n2;
  4’d5: M <= M_n2;
  4’d6: M <= M_n2;
  4’d7: M <= M_n2;
  4’d8: M <= M_n3;
  4’d9: M <= M_n3;
  4’d10:  M <= M_n3;
  4’d11:  M <= M_n3;
  4’d12:  M <= M_n2;
  4’d13:  M <= M_n2;
  4’d14:  M <= M_n2;
  4’d15:  M <= M_n2;
  endcase
  end
//--------------------------------------------------------

//--------Reading sprite ROM address into a local reg when sprite flag is ON-------

  // address of sprite rom blocks
  assign addr = (ninja)? (regiony*64+regionx):12’d0;  //ninja
  assign addr_b1 = (book1)? (regiony1*64+regionx1):12’d0; // 6 moving sprites
  assign addr_b2 = (book2)? (regiony2*64+regionx2):12’d0;
  assign addr_b3 = (book3)? (regiony3*64+regionx3):12’d0;
  assign addr_b4 = (book4)? (regiony4*64+regionx4):12’d0;
  assign addr_b5 = (book5)? (regiony5*64+regionx5):12’d0;
  assign addr_b6 = (book6)? (regiony6*64+regionx6):12'd0;

  assign addr_dp = (dip_fl)? (dipy*64+dipx):12'd0;    //diploma sprite

  assign addr_s1 = (stage1)? (stagey1*64+stagex1):12'd0; //3 stages
  assign addr_s2 = (stage2)? (stagey2*64+stagex2):12'd0;
  assign addr_s3 = (stage3)? (stagey3*64+stagex3):12'd0;

  assign addr_ps = (pass_fl)? (passy*64+passx):12'd0; //pass/fail
  assign addr_fl = (fail_fl)? (faily*64+failx):12'd0;

  assign addr_t = (try_fl || sun_fl || moon_fl || rain_fl)? (tryy*64+tryx):12'd0; //sun/moon
    /rain/tryagain

  assign addr_sym = (sym_fl)? (symy*64+symx):12'd0; //symbol
  assign addr_nun = (nun_fl)? (nuny*400+nunx%400):12'd0; //nuny name

  assign addr_nl1 = (nin1_fl)? (nin1y*32+nin1x):12'd0; //3 ninja lives
  assign addr_nl2 = (nin2_fl)? (nin2y*32+nin2x):12'd0;
  assign addr_nl3 = (nin3_fl)? (nin3y*32+nin3x):12'd0;

  assign addr_bg = (skyline)? ((yrow-155)*160+xrow%160):15'd0; //background sprite

//which of the three ninja lives address
    always @(*) begin
        if (nin1_fl)
            addr_nl = addr_nl1;
        else if (nin2_fl)
            addr_nl = addr_nl2;
        else if (nin3_fl)
            addr_nl = addr_nl3;
        else
            addr_nl = 12'd0;
    end

    // assign number ROM addresses based on the number in ones/tens/hundreds place
    always_comb begin
        if (one_fl) begin
            temp3 = one;
            temp_fl = 1;
            tempx = onex;
            tempy = oney;
        end
        else if (ten_fl) begin
            temp3 = ten;
            temp_fl = 1;
            tempx = tenx;
            tempy = teny;
        end
        else if (hun_fl) begin
            temp3 = hun;
            temp_fl = 1;
            tempx = hunx;
            tempy = huny;
        end
        else begin
            temp_fl = 0;
            temp3 = 0;
            tempx = onex;
            tempy = oney;
        end

        addr_sc0 = (temp_fl == 1 && temp3 == 0)?(tempy*32+tempx):12'd0;
        addr_sc1 = (temp_fl == 1 && temp3 == 1)?(tempy*32+tempx):12'd0;
        addr_sc2 = (temp_fl == 1 && temp3 == 2)?(tempy*32+tempx):12'd0;
        addr_sc3 = (temp_fl == 1 && temp3 == 3)?(tempy*32+tempx):12'd0;
        addr_sc4 = (temp_fl == 1 && temp3 == 4)?(tempy*32+tempx):12'd0;
        addr_sc5 = (temp_fl == 1 && temp3 == 5)?(tempy*32+tempx):12'd0;
```

```verilog
        addr_sc6 = (temp_fl == 1 && temp3 == 6)?(tempy*32+tempx):12'd0;
        addr_sc7 = (temp_fl == 1 && temp3 == 7)?(tempy*32+tempx):12'd0;
        addr_sc8 = (temp_fl == 1 && temp3 == 8)?(tempy*32+tempx):12'd0;
        addr_sc9 = (temp_fl == 1 && temp3 == 9)?(tempy*32+tempx):12'd0;

    end

//Since only one address used for all the numbers ROM blocks, select which address based on
// the number in the ones/tens/hundreds place
    always @(*) begin
        if (one_fl)
            temp2 = one;
        else if (ten_fl)
            temp2 = ten;
        else if (hun_fl)
            temp2 = hun;
        else
            temp2 = one;

        case (temp2)
            4'd0: addr_sc = addr_sc0;
            4'd1: addr_sc = addr_sc1;
            4'd2: addr_sc = addr_sc2;
            4'd3: addr_sc = addr_sc3;
            4'd4: addr_sc = addr_sc4;
            4'd5: addr_sc = addr_sc5;
            4'd6: addr_sc = addr_sc6;
            4'd7: addr_sc = addr_sc7;
            4'd8: addr_sc = addr_sc8;
            4'd9: addr_sc = addr_sc9;
            default: addr_sc = addr_sc0;
        endcase
    end

// stage/pass/fail/diploma sprites have same address, selecting here based on flag
  always @(*) begin
    if (stage1 )
      addr_s = addr_s1;
    else if (stage2 )
      addr_s = addr_s2;
    else if (stage3 )
      addr_s = addr_s3;
        else if (pass_fl)
            addr_s = addr_ps;
        else if (fail_fl)
            addr_s = addr_fl;
        else if (dip_fl)
            addr_s = addr_dp;
    else addr_s = 12'd0;
  end
//------------------------------------------------------------------------------------

//-----------Writing sprite data to buffers at clock edge----------------------------

  // counter for moving ninja sword on position
  always@(vcount)
  if (vcount == 520) begin
  cnt <= cnt + 1;
  end
  else begin
  cnt <= cnt;
  end

  //counter for writing into the buffers
  always@(posedge vcount[0])
  buf_cnt <= buf_cnt + 1;

  // writing into the buffers
  always @(posedge clk) begin
    if (buf_cnt==0)
        buffer1[xrow] <= M_buf;
    else
        buffer2[xrow] <= M_buf;
  end

  always @(posedge clk) begin
    if (buf_cnt==0)
        line_buffer[xrow] <= buffer2[xrow];
    else
        line_buffer[xrow] <= buffer1[xrow];
  end


    //----------------------Sprite priority encoder---------------------
   always_comb begin
        M_buf = 12'h0fe;    // write white to pixel bt default

    if (ninja==1 && M!=12'd4095) begin
        M_buf = M;
    end
    else if ((book==1)  && M_b!=12'd4095) begin
      M_buf = M_b;
    end
    else if (((nin1_fl) || (nin2_fl) || (nin3_fl)) && M_nl!=12'd4095) begin
      M_buf = M_nl;
    end
    else if ((one_fl || ten_fl || hun_fl) && M_sc!=12'd4095) begin
        M_buf = M_sc;
    end
    else if ((sun_fl || moon_fl || try_fl || rain_fl ) && (M_temp != 12'd4095 && M_temp != 12'd0)) begin
        M_buf = M_temp;
    end
    else if ((nun_fl) && M_nun!=12'd4095) begin
        M_buf = M_nun;
    end
    else if ((sym_fl) && M_sym!=12'd4095) begin
        M_buf = M_sym;
    end
    else if ((stage1 || stage2 || stage3 ||pass_fl ||fail_fl || dip_fl)&& M_s!=12'd4095) begin
      M_buf = M_s;
    end
     else if ((skyline==1) && (M_bg!=12'd4095)) begin
      M_buf = M_bg;
      end
    else if (sky==1) begin
    M_buf = 12'h0fe;
    end

    else if (black==1) begin
        M_buf = 12'h000;
      end
    end
//----------------------------------------------------------------------------------

//-----------------------Writing RGB values----------------------------------
assign VGA_R = {line_buffer[xrow][11:8],line_buffer[xrow][11:8]};
assign VGA_G = {line_buffer[xrow][7:4],line_buffer[xrow][7:4]};
assign VGA_B = {line_buffer[xrow][3:0],line_buffer[xrow][3:0]};

endmodule
```

../hardware_cleaned/RGB_controller.sv

```
/*
```

```
 * VGA LED emulator
 *
 * Stephen A. Edwards, Columbia University
 * Modified as needed
 */

module VGA_LED_Emulator(clk50, reset, hcount, vcount,
                VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);
 input wire clk50, reset;
 output wire [10:0] hcount;
 output wire [9:0]  vcount;
 output wire 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          |____|
 */

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

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

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

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

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

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

endmodule // VGA_LED_Emulator
```

../hardware_cleaned/VGA_LED_Emulator.sv

```
/* VGA_LED.sv
top-level module for VGA display, contains instantiations for sprite ROM blocks
and also communicates with the avalon bus*/

module VGA_LED(
       //read from avalon bus
       input logic clk,
         input logic    reset,
         input logic [15:0]  writedata,
         input logic    write,
         input      chipselect,
       input logic [3:0] address,
       // output to VGA
         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);


  //-------- coordinates of the sprites read from software------------
    logic [15:0] x,y;//ninja coordinates
  logic [15:0] x1,y1;//book1 coordinates
  logic [15:0] x2,y2;//book2 coordinates
  logic [15:0] x3,y3;//book3 coordinates
  logic [15:0] x4,y4;//book4 coordinates
  logic [15:0] x5,y5;//bomb coordinates
    //------------------------------------------------------------------

  //-----------address of sprite block roms---------------------
  wire [11:0] addr;       // ninja address
  wire [11:0] addr_b1;  // book1 address
  wire [11:0] addr_b2;  // book2 address
  wire [11:0] addr_b3;  // book3 address
  wire [11:0] addr_b4;  // book4 address
  wire [11:0] addr_b5;  // bomb address
  wire [11:0] addr_b6;  // bomb address
  wire [11:0] addr_s;     // stage address
  wire [11:0] addr_ps;  // pass address
  wire [11:0] addr_fl;  // fail address
  wire [14:0] addr_bg;  // background address
  wire [11:0] addr_sc;  // score address
    wire [11:0] addr_nl;    // life address
    wire [14:0] addr_nun;   //Ninja University name address
    wire [14:0] addr_sym;   //NUNY symbol address
    wire [14:0] addr_t;     //Address for sun/moon/rain sprites, used for different levels
    //-----------------------------------------------------------------

  //-------------- sprite block rom data (12 bits)------------
  wire [11:0] M_bg1;  //Data for background split sprite 1
  wire [11:0] M_bg2;  //Data for background split sprite 2
  wire [11:0] M_bg3;  //Data for background split sprite 3
  wire [11:0] M_bg4;  //Data for background split sprite 4
  wire [11:0] M_n1;   //Data for ninja (sword position 1)
  wire [11:0] M_n2;   //Data for ninja (sword position 2)
  wire [11:0] M_n3;   //Data for ninja (sword position 3)
  wire [11:0] M_b1;   //Data for book1 sprite object
  wire [11:0] M_b2;   //Data for book2 sprite object
  wire [11:0] M_b3;   //Data for book3 sprite object
  wire [11:0] M_b4;   //Data for book4 sprite object
  wire [11:0] M_b5;   //Data for book5 sprite object
  wire [11:0] M_b6;   //Data for book6 sprite object
  wire [11:0] M_s1;   //Data for stage1 sprite
  wire [11:0] M_s2;   //Data for stage2 sprite
  wire [11:0] M_s3;   //Data for stage3 sprite
  wire [11:0] M_ps;   //Data for pass sprite
  wire [11:0] M_dp;   //Data for diploma sprite
  wire [11:0] M_fl;   //Data for fail sprite
    wire [11:0] M_sc0;  //Data for number 0 in score
    wire [11:0] M_sc1;  //Data for number 1 in score
    wire [11:0] M_sc2;  //Data for number 2 in score
    wire [11:0] M_sc3;  //Data for number 3 in score
    wire [11:0] M_sc4;  //Data for number 4 in score
    wire [11:0] M_sc5;  //Data for number 5 in score
    wire [11:0] M_sc6;  //Data for number 6 in score
    wire [11:0] M_sc7;  //Data for number 7 in score
    wire [11:0] M_sc8;  //Data for number 8 in score
    wire [11:0] M_sc9;  //Data for number 9 in score

    wire [11:0] M_nl1;  //Data for ninja life 1
    wire [11:0] M_nl2;  //Data for ninja life 2
    wire [11:0] M_nl3;  //Data for ninja life 3

    wire[11:0] M_sun;   //Data for sun sprite for level 1
    wire[11:0] M_mn;    //Data for moon sprite for level 2
    wire[11:0] M_rn;    //Data for rain sprite for level 3

    wire[11:0] M_try;   //Try again sprite used in last screen

    wire[11:0] M_nun;   //Data for NUNY name sprite on selection screen
    wire[11:0] M_sym;   //Data for NUNY symbol sprite
    //--------------------------------------------------------------

    //-------------------Misc declarations-------------------------

    wire [10:0] hcount; //hcount for VGA
  wire [9:0] vcount;  //vcount for VGA
  wire [10:0] xrow;   //Reading the horizontal axis of display
  wire [1:0] state;   //state read from SW to decide which screen
  wire [2:0] screen;  //which screen (1-hot code)
  wire [2:0] level;   //which level BA, MS, Phd (1-hot code)
  wire result;    //result pass or fail

    wire [7:0] score; //What is the score read from SW
    wire [3:0] one; //One's place of score
    wire [3:0] ten; //Ten's place of score
    wire [3:0] hun; //Hundred's place of score

    reg [2:0] nin_life = 3'b000; //How many ninja lives to display
    //-------------------------------------------------------------

    //-------------Call VGA controller--------------------------
  VGA_LED_Emulator led_emulator(.clk50(clk),
                      .reset(reset),
                      .hcount(hcount),
                      .vcount(vcount),
                      .VGA_CLK (VGA_CLK),
                      .VGA_HS (VGA_HS),
                      .VGA_VS (VGA_VS),
                      .VGA_BLANK_n (VGA_BLANK_n),
                      .VGA_SYNC_n (VGA_SYNC_n));
    //---------------------------------------------------------------

  //--------------block rom for sprites---------------------------
  ninja1 ninja1(.clock(VGA_CLK), .address(addr), .q(M_n1));   //ninja sword position 1
  ninja2 ninja2(.clock(VGA_CLK), .address(addr), .q(M_n2));   //ninja sword position 2
```

```
ninja3 ninja3(.clock(VGA_CLK), .address(addr), .q(M_n3));   //ninja sword position 3

reading book1(.clock(VGA_CLK), .address(addr_b1), .q(M_b1));    //reading sprite
exam book2(.clock(VGA_CLK), .address(addr_b2), .q(M_b2));       //exam sprite
homework book3(.clock(VGA_CLK), .address(addr_b3), .q(M_b3));   //homework sprite
bomb book4(.clock(VGA_CLK), .address(addr_b4), .q(M_b4));   //Bomb sprite
pizza book5(.clock(VGA_CLK), .address(addr_b5), .q(M_b5));  //Pizza sprite
thesis_new book6(.clock(VGA_CLK), .address(addr_b6), .q(M_b6)); //Thesis sprite

bg1_new  prom_bg1(.clock(VGA_CLK), .address(addr_bg), .q(M_bg1));   //Background split 1
  sprite
bg2_new  prom_bg2(.clock(VGA_CLK), .address(addr_bg), .q(M_bg2));   //Background split 2
  sprite
bg3_new  prom_bg3(.clock(VGA_CLK), .address(addr_bg), .q(M_bg3));   //Background split 3
  sprite
bg4_new  prom_bg4(.clock(VGA_CLK), .address(addr_bg), .q(M_bg4));   //Background split 4
  sprite

bach_new level1(.clock(VGA_CLK), .address(addr_s), .q(M_s1));   //BA level sprite
mast_new level2(.clock(VGA_CLK), .address(addr_s), .q(M_s2));   //MA level sprite
phd_new level3(.clock(VGA_CLK), .address(addr_s), .q(M_s3));    //PhD level sprite

  pass_new ps(.clock(VGA_CLK), .address(addr_s), .q(M_ps));   //pass sprite
  fail_new fl(.clock(VGA_CLK), .address(addr_s), .q(M_fl));   //fail sprite
diploma_new dip0(.clock(VGA_CLK), .address(addr_s), .q(M_dp));  //diploma sprite

  zero_new2 sc0(.clock(VGA_CLK), .address(addr_sc), .q(M_sc0));   //Zero number sprite
  one_new2 sc1(.clock(VGA_CLK), .address(addr_sc), .q(M_sc1));    //One number sprite
  two_new2 sc2(.clock(VGA_CLK), .address(addr_sc), .q(M_sc2));    //Two number sprite
  three_new2 sc3(.clock(VGA_CLK), .address(addr_sc), .q(M_sc3));  //Three number sprite
  four_new2 sc4(.clock(VGA_CLK), .address(addr_sc), .q(M_sc4));   //Four number sprite
  five_new2 sc5(.clock(VGA_CLK), .address(addr_sc), .q(M_sc5));   //Five number sprite
  six_new2 sc6(.clock(VGA_CLK), .address(addr_sc), .q(M_sc6));    //Six number sprite
  seven_new2 sc7(.clock(VGA_CLK), .address(addr_sc), .q(M_sc7));  //Seven number sprite
  eight_new2 sc8(.clock(VGA_CLK), .address(addr_sc), .q(M_sc8));  //Eight number sprite
  nine_new2 sc9(.clock(VGA_CLK), .address(addr_sc), .q(M_sc9));   //Nine number sprite

life_new nl1(.clock(VGA_CLK), .address(addr_nl), .q(M_nl1));    //Life sprite instantiated
   3 times
life_new nl2(.clock(VGA_CLK), .address(addr_nl), .q(M_nl2));
life_new nl3(.clock(VGA_CLK), .address(addr_nl), .q(M_nl3));

sun sun0(.clock(VGA_CLK), .address(addr_t), .q(M_sun)); // Sun sprite in BA level
moon mn0(.clock(VGA_CLK), .address(addr_t), .q(M_mn));  //Moon sprite in MA level
rain rn0(.clock(VGA_CLK), .address(addr_t), .q(M_rn));  //Rain sprite in PhD level

nuny_new2 nun0(.clock(VGA_CLK), .address(addr_nun), .q(M_nun)); //NUNY name sprite used in
   selection sprite
ninjasymbol sym0(.clock(VGA_CLK), .address(addr_sym), .q(M_sym));   //Ninja symbol sprite

tryagain try0(.clock(VGA_CLK), .address(addr_t), .q(M_try));    //Try again sprite
  //--------------------------------------------------------------------

  //----------Call the sprite controller module-----------------------------------
RGB_controller controller_1(.clk(VGA_CLK),
                  .clk50(clk),
                  .hcount(hcount),
                  .vcount(vcount),
                  .x(x), .y(y),
                  .x1(x1), .y1(y1),
                  .x2(x2), .y2(y2),
                  .x3(x3), .y3(y3),
                  .x4(x4), .y4(y4),
                  .x5(x5), .y5(y5),
                                      .one(one),
                                      .ten(ten),
                                      .hun(hun),
                  .addr(addr),
                    .addr_b1(addr_b1),
                    .addr_b2(addr_b2),
                    .addr_b3(addr_b3),
                    .addr_b4(addr_b4),
                    .addr_b5(addr_b5),
                    .addr_b6(addr_b6),
                    .addr_s(addr_s),
                    .addr_sc(addr_sc),
                    .addr_nl(addr_nl),
                    .addr_t(addr_t),
                    //.addr_sun(addr_sun),
                    //.addr_mn(addr_mn),
                    //.addr_rn(addr_rn),
                    //.addr_try(addr_try),
                    .addr_nun(addr_nun),
                    .addr_sym(addr_sym),
                    .M_s1(M_s1),
                    .M_s2(M_s2),
                    .M_s3(M_s3),
                    .M_n1(M_n1),
                    .M_n2(M_n2),
                    .M_n3(M_n3),
                    .M_b1(M_b1),
                    .M_b2(M_b2),
                    .M_b3(M_b3),
                    .M_b4(M_b4),
                    .M_b5(M_b5),
                    .M_b6(M_b6),
                    .M_ps(M_ps),
                    .M_dp(M_dp),
                    .M_fl(M_fl),
                                        .M_sc0(M_sc0),
                                        .M_sc1(M_sc1),
                                        .M_sc2(M_sc2),
                                        .M_sc3(M_sc3),
                                        .M_sc4(M_sc4),
                                        .M_sc5(M_sc5),
                                        .M_sc6(M_sc6),
                                        .M_sc7(M_sc7),
                                        .M_sc8(M_sc8),
                                        .M_sc9(M_sc9),
                                        .M_nl1(M_nl1),
                                        .M_nl2(M_nl2),
                                        .M_nl3(M_nl3),
                                        .M_sun(M_sun),
                                        .M_mn(M_mn),
                                        .M_rn(M_rn),
                                        .M_try(M_try),
                                        .M_nun(M_nun),
                                        .M_sym(M_sym),
                    .addr_bg(addr_bg),
                    .M_bg1(M_bg1),
                    .M_bg2(M_bg2),
                    .M_bg3(M_bg3),
                    .M_bg4(M_bg4),
                    .screen(screen),
                                        .level(level),
                                        .result(result),
                    .nin_life(nin_life),
                    //.line_buffer(line_buffer)
                    .VGA_R(VGA_R),
          .VGA_G(VGA_G),
          .VGA_B(VGA_B)
                    );
//---------------------------------------------------------------------------


   //---------------Read from the VGA peripheral memory from various addresses--------
   always_ff @(posedge clk)
     if (reset) begin
  x <= 16'd300;
  y <= 16'd200;
  x1 <= 16'd10;
  y1 <= 16'd300;
  x2 <= 16'd70;
  y2 <= 16'd300;
  x3 <= 16'd200;
  y3 <= 16'd300;
  x4 <= 16'd300;
  y4 <= 16'd300;
  x5 <= 16'd500;
  y5 <= 16'd300;
  state <= 2'b00;
  score <= 8'b0;
  nin_life <= 3'b0;
  level <= 3'b0;
  result <= 1'b0;

  end
  else if (chipselect && write)
  begin

  case(address)
  4'b0000: x <= writedata;    //Get coordinates of ninja and other moving sprites
  4'b0001: y <= writedata;
  4'b0010: x1 <= writedata;
  4'b0011: y1 <= writedata;
  4'b0100: x2 <= writedata;
  4'b0101: y2 <= writedata;
  4'b0110: x3 <= writedata;
  4'b0111: y3 <= writedata;
  4'b1000: x4 <= writedata;
  4'b1001: y4 <= writedata;
  4'b1010: x5 <= writedata;
  4'b1011: y5 <= writedata;
  4'b1100:begin           //get screen, level, pass/fail info
            state <= writedata[1:0];
            level <= writedata[4:2];
            result <= writedata[5];
        end
  4'b1101: score <= writedata[7:0];   //get score
  4'b1110: nin_life <= writedata[2:0];    //get lives remaining
  4'b1111: state<= 2'b0;  //default
  endcase
  end
//----------------------------------------------------------------------------------

//-------------Select screen based on the state read from SW-----------------------
  always_ff @(posedge clk) begin
  if (reset)
    screen = 3'b010;
  else case(state)
    2'b00: screen <= 3'b010;
    2'b01: screen <= 3'b001;
    2'b10: screen <= 3'b100;
    default: screen <=3'b010;
    endcase
  end
//---------------------------------------------------------------------------------


    //--------------Decimal to BCD converter to convert score into ones/tens/hundreds
    --------
    integer i;
    always @(score) begin
        hun = 4'd0;
    ten = 4'd0;
    one = 4'd0;

    for (i = 7; i >= 0; i = i -1) begin
      if (hun >= 5)
        hun = hun + 3;
      if (ten >= 5)
        ten = ten + 3;
      if (one >= 5)
        one = one + 3;

      hun = hun << 1;
      hun[0] = ten[3];
      ten = ten << 1;
      ten[0] = one[3];
      one = one << 1;
      one[0] = score[i];
    end
  end
 //-------------------------------------------------------------------------------------
endmodule
```

../hardware_cleaned/VGA_LED.sv