CSEE4840 Embedded Systems Final Report

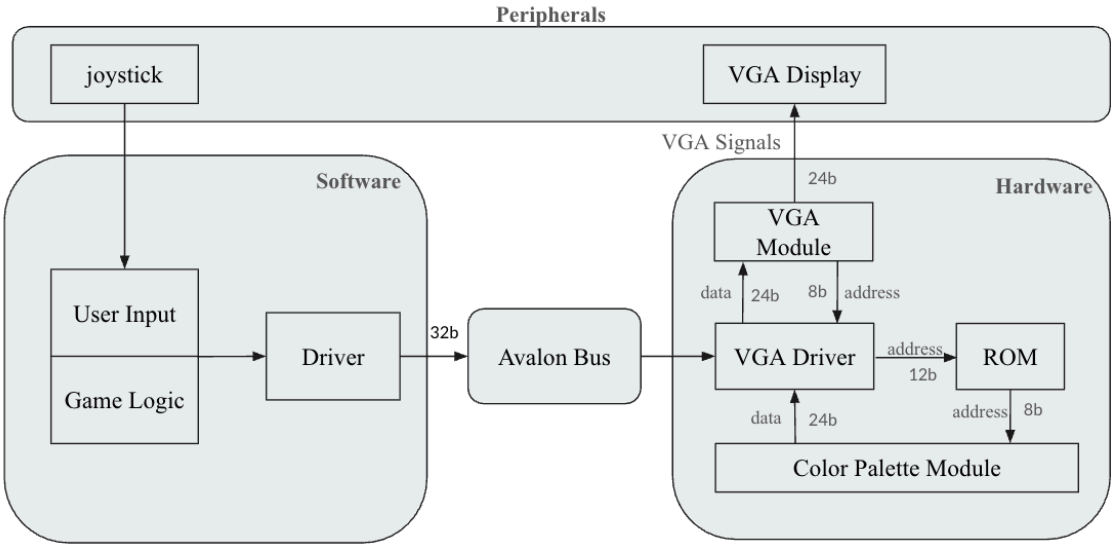
Spaceship Defender Game

`

Mingzhi Li (ml5160), Noah Hartzfeld (nah2178), Hiroki Endo (he2305)

Jingyi Lai (jl6932), Zhengtao Hu (zh2651)

1. System Overview



We used both the ARM CPU and FPGA on the DE1-SoC platform.

Controllers with joystick and VGA Monitor are used as the input and output peripherals.

On the software (CPU) side, libusb is used to capture input from a USB game controller. The input is processed within the game logic and corresponding video/audio commands are delivered to the Avalon bus via custom kernel drivers.

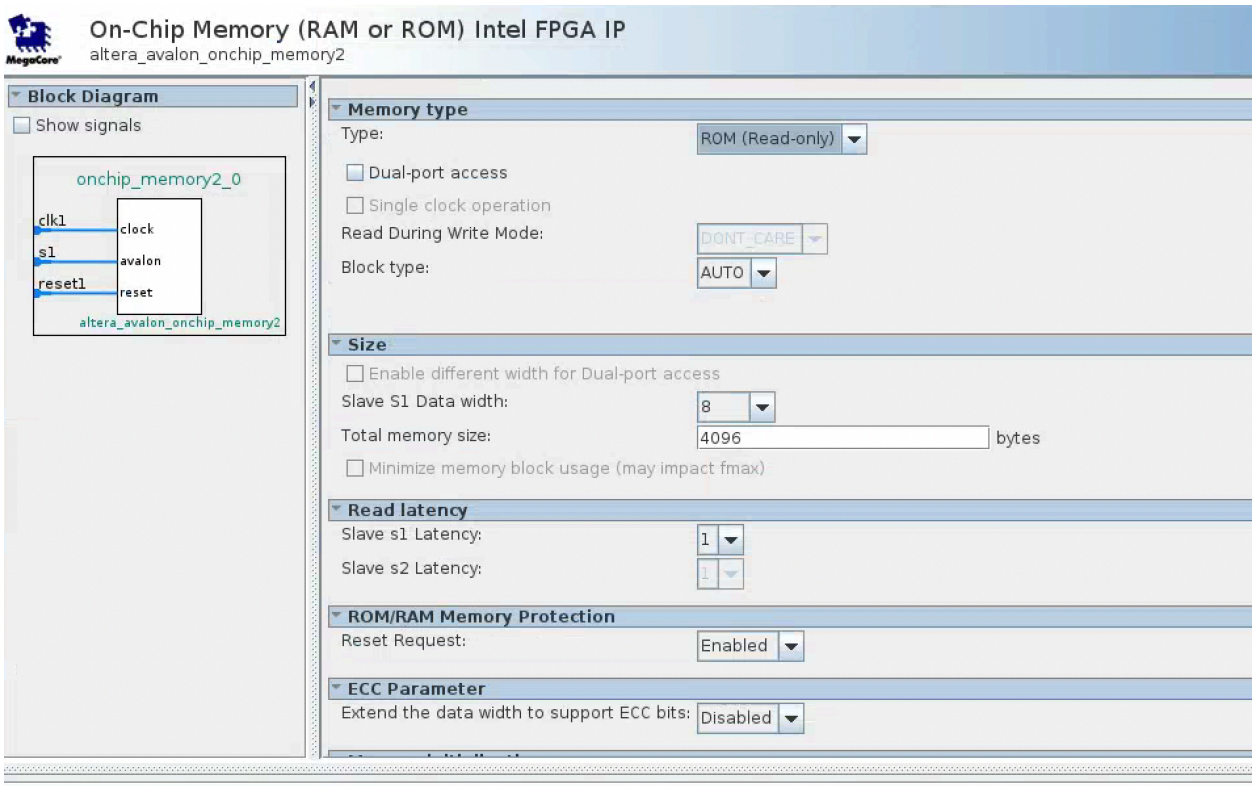
The hardware and software are connected via the Avalon Bus Interface.

On the hardware (FPGA) side, we implemented sprite and tile rendering modules to output graphics to the screen through the VGA interface.

2. Hardware Design

2.1. Graphics display

The primary hardware algorithm handles the graphical display logic. The graphical sprites we employed are stored in on-chip memory ROMs that are created and set up using on-chip system memory IP blocks within Platform Designer.



We converted the .png sprite images into .mif memory initialization files for ROM population.

```
WIDTH=8;
DEPTH=4096;

ADDRESS_RADIX=HEX;
DATA_RADIX=HEX;

CONTENT BEGIN
    0000 : 00;
    0001 : 00;
    0002 : 00;
    0003 : 00;
    0004 : 00;
    0005 : 00;
    0006 : 00;
    0007 : 00;
    0008 : D2;
    0009 : 00;
```

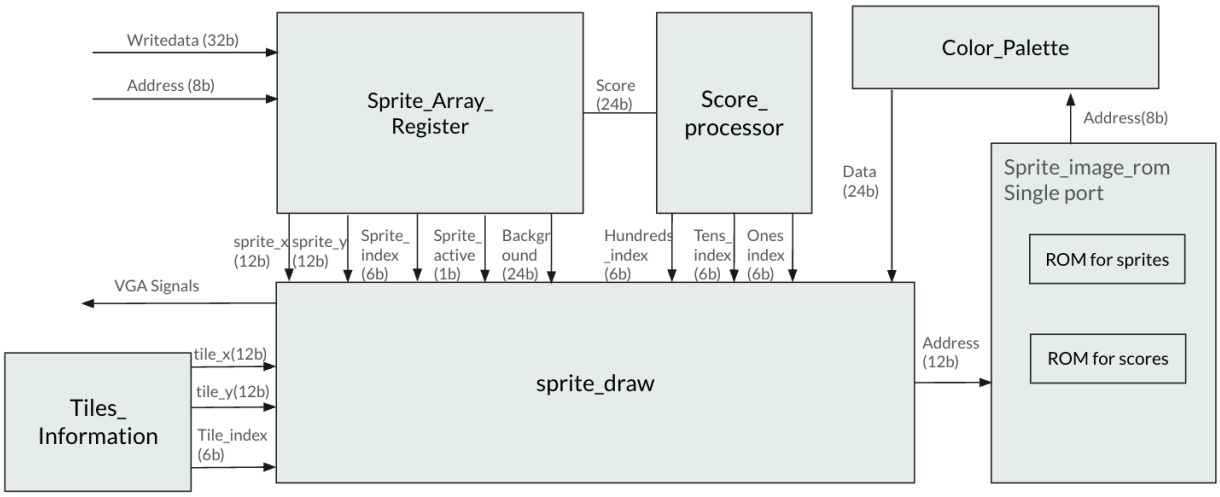
To display graphics, our system determines ROM read addresses based on ioctl writes from the device driver and the screen position. The ROM values (defined by .mif files) are hex values for each pixel in the image, with each pixel corresponding to a memory address. These output hex values are used to look up values for VGA RGB signals.

The color determination is handled by the build_color_palette() function, which creates a 256-color palette. First, it generates 216 base colors using triple nested loops that create 6 intensity levels each for red, green, and blue components (multiplied by 51). Then it fills the remaining 40 colors using mathematical formulas with different multipliers (47, 91, and 137). This complete palette is used to determine which specific color from the available 256 options should be displayed at each screen location.

2.2 Sprite and tile drawing

2.2.1 Workflow

We encapsulate everything needed to display graphics to the vga_top module. This includes 3 major components: sprite array register, score processor and sprite drawing logic.



The figure shows the workflow of sprites and tiles.

First of all, we treat the lower 24 bits of the data at address with index 0 as the background. The 24 bits of data represents the RGB information of the gameplay background color, with 8 bits allocated to each channel.

Then, we define the lower 16 bits of the data at address with index 1 as the gameplay scoring information. It is enough for our three levels' scoring.

Starting at address index 2, the Sprite Array Register parses each 32-bit word received from software into sprite fields: bits 31–20 become the X coordinate, bits 19–8 the Y coordinate, bits 7–2 the sprite index, and bit 1 the "active" flag (and bit 0 is reserved).

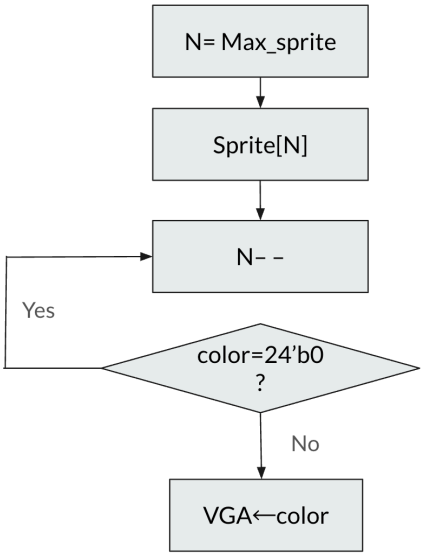
Once unpacked, these fields fan out to both the Score Processor and the Sprite Drawing logic. The Score Processor takes its input and computes three separate 6-bit values—hundreds, tens, and ones—for the on-screen score, then forwards those indices on to the Sprite Drawing logic as well.

To render both tiles and sprites, the hardware needs each object's position plus the address of its pixel data in ROM. Tile parameters are hard-wired inside the VGA Ball module, whereas sprite parameters come dynamically from the Sprite Array Register. Score‑tile positions are fixed but their palette indices originate from the Score Processor.

With all data received, the Sprite Drawing logic steps through every pixel. It determines which sprite (or tile) covers the current X/Y, fetches the corresponding palette index from ROM, then looks up the 24-bit RGB value in the palette and drives the VGA_R/G/B outputs accordingly.

2.2.2 Overlap and transparency

The sprite transparency implementation utilizes a color-key approach where pure black pixels (RGB value 24'h0) serve as transparent regions. During the VGA rendering process, the controller performs a pixel-by-pixel evaluation, checking each sprite pixel against the transparency threshold. When a pixel matches the black transparency key, the system skips rendering that pixel and proceeds to display either the next sprite in the rendering pipeline or the background color. For non-transparent pixels, the controller renders the sprite's actual color value normally. This implementation enables natural sprite overlap without visible rectangular borders while maintaining computational efficiency through simple color comparison operations. The resulting graphics system successfully supports complex layered visuals where multiple sprites can be displayed with seamless transparency effects, significantly enhancing the overall visual quality of the rendered output.



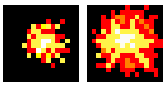
2.2.3 Sprite registers

We design the address to 7 bits, and set the max of the number of sprites to 100, so hardware contains 100 registers, each used to store the positional, image index and the activeness of a sprite. Each register is 32 bits wide.The address distribution is shown as below, we use the first register for rgb information of background (24 bits), and second address for score (8bit). For the rest address, each data will contain a group of

information for a sprite. We define the priority in the screen from software, the lower the index of address is, the priorier the sprite shows in the screen.

address	data	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	background	\								background_r								background_g								background_b							
1	score																	Score															
[127:2]	sprites	sprite_x												sprite_y												sprite_index						active	\

2.3. Memory

Category	Graphics	Size (pixel)	# of images	Total Size (byte)
Spaceship		16*16	3	768
Fire		16*16	1	256
Enemy Bullet		16*16	3	768
Enemy Ship		16*16	3	768
Player Bullet		16*16	1	256
Power Ups		16*16	3	768
Explosion Effect		16*16	2	512
Numbers		16*16	10	2560
Score		16*16	5	1280

The visual sprites used in the game are stored in two ROMs.

All the sprites used are with dimensions of 16x16 pixels, totaling 256 pixels. Each pixel uses 8 bit or 1 byte palette, and allows up to 256 possible defined colors. Multiplying yields a size usage of 256 bytes for a single sprite. Sprites of Spaceship, Fire, ENemy Bullet, Enemy Ship, Player Bullet, Power Ups, and Explosion Effects (a total of 16 sprites) are stored in the first ROM, occupying a total of 4096 Bytes, at 4096 lines with width of 8 bits. Sprites of Numbers and SCORE are stored in the second ROM, occupying a total of 3840 Bytes, also at 3840 lines with width of 8 bits.

2.4. Attempts

Some features and/or functions were attempted but were not successful in making it into the final game.

One of such attempts is the implementation of Linear Feedback Shift Register (LSFR) to generate 16 bit random sequences that could be used to produce the position and flickering state of the stars in the gameplay background and control other mechanics of the game such as PowerUp drops. In hardware code, 16-bit register vectors were added and synthesized to 16 D Flip-Flops and chained as a Fibonaccis LFSR. In the reset edge, it is seeded to a non-zero constant (in our case we chose 1), and on each frame-start (which is when “hcount” and “ycount” becomes 0), an always_ff modules will shift into a new feedback but computed by the dedicated polynomial. The taps are connected through XOR gates and shift from the maximal-period sequence and giving a pseudo random bit for every run of LSFR. In the render logic, we tried to put it at coordinates generated by the LSFR and flicker according to the LSFR.

Another attempt was the usage of audio. We connected the FPGA’s Wolfson WM8731 output to a speaker and used the AudioCodecDriver IPs to configure the CODEC, and built a CODEC interface with two 16-bit DAC registers that fetched samples from the Audio ROM IP we implemented and stored the MIF files from pre-recorded WAV sounds. We set up the Avalon Bus interface with connections to the ROMS, however no sound was ever heard from the speaker, most likely because the interface was not initialized correctly.

3. Software Design

3.1 Game Logic

3.1.1 Overview

This is a single-player game. The player controls a spaceship object on the screen using a Gamepad controller. Kill every enemy in each of three increasingly difficult rounds, and accumulate as many points as possible. In each game, the player begins with five lives and can gain points by killing enemies, but will lose points for each life they lose. Player lives are displayed at the bottom of the screen, represented by ships. The score is displayed at the top of the screen, and power-ups are displayed at the bottom of the screen when active.

3.1.2 Object Structures

The basic structure of each object contains an x and a y coordinate, an x and a y velocity, a sprite index, and an active bit.

- X and Y coordinates
    - Represent the location of the object on the screen
    - Communicates to the hardware where to write the sprite on the screen

- Active Bit
    - Whether an object should be displayed on the screen
    - The hardware only displays the object sprite if the active bit is flipped

- Sprite Index
    - Corresponds to the index into the ROM array for that object's sprite
    - The hardware references the index to know where to look in ROM when drawing
    - The sprites can be easily changed to depict object movement
        - Display the direction of movement of the ship
        - The ship's flames are only present when it moves forward
        - On death, the ship and enemy sprites flash the explosion graphics

3.1.3 Ship Mechanics

The ship can move freely around the screen, including diagonally. The user can hold down on the d-pad to continuously move in the correct direction.

The main bullet-firing mechanic allows the ship to shoot a single bullet at a time. Another bullet is not available until the fired one has struck an enemy or traveled off the screen. The user must release the button to fire a bullet; during the bullet powerup, the user cannot shoot rapid fire.

3.1.4 Enemies Generation and Movement

On initialization, the program will generate rows of enemies in the upper half of the screen. There are three types of enemies. Each row contains only one type of enemy.

There are three types of enemies worth varying amounts of points.

- Enemy 1
    - Does not fire bullets
    - Chases the user around the screen for a couple of seconds
- Enemy 2
    - Small and fast
    - Hovers over the ship, firing straight down
- Enemy 3
    - Slow, large zig-zags down the screen
    - Fires directional shots towards the ship

Enemies in row shuffle back and forth together, as a large cluster. Enemies descend on the ship individually, but frequently enough, so that many overlap and attack at the same time. As the round time increases, there are numerous points where multiple enemies leave their rows at the same time. Depending on the round, the frequency of enemy attacks increases.

If an enemy runs into the ship, both the enemy and the ship will die, but points are not awarded for killing an enemy; only points are deducted for losing a life. If an enemy's bullet hits the ship, only the ship will die, and the enemy will return to its row. The ship will not respawn until all enemies have returned to their row. If all enemies on the screen are killed, a new round will spawn with an increased number of enemies for three rounds.

Enemy bullets are held in an array that each can pull from. This allows us to keep the number of preallocated objects down, as only attacking enemies need bullets, not the ones still in rows.

3.1.5 Power-ups

We have three separate kinds of power-ups that are dropped randomly after 15 enemy kills. The power-ups are generated opposite the ship and slowly descend off the screen. If the user fails to acquire the power-up, they will not need to kill another 15 enemies, but only a few more to produce another one. If the ship loses a life with an active power-up, it will lose it. If the ship completes a round with an active power-up, it carries over to the next. The power-ups provide either an extra life, a limited period of quicker movement speed, or a period where the player can shoot three bullets at a time rather than only one. If the user has not lost a life, an extra life power-up will not be dropped.

3.2 Collision Detection

Our game uses the 640 x 480 screen as an interface. All objects are displayed as 16 x 16 pixel sprites. According to our game logic, a collision occurs when an enemy's bullet hits the ship, a ship’s bullet hits an enemy, or an enemy hits the ship. To be specific, the coordinates of the top left corners of the two objects will be compared, and if their horizontal and vertical distances are smaller than or equal to 16 (the ship/enemies’ height/width), they will be seen as colliding.

The same detection happens when the player is reaching the boundaries of the screen. For right and lower bounds in specific, we compare if the object’s horizontal/vertical position is less than the difference between the screen’s width/height and the object’s width/height. This is an effective detection since it ensures the sprites' top left corner will never move beyond the allowable range, guaranteeing their full width and height never go off screen.

When a bullet collides with an enemy, the bullet is set inactive, and the enemy sprite is changed to a series of explosion sprites. The same occurs if the ship collides with an enemy bullet or when the ship collides with an enemy.

When the ship collides with the screen boundary, the program will not allow it to move further in that direction, so that the ship sprite is always completely visible.

When the ship collides with the powerup, the powerup position will be moved to the bottom of the screen to indicate that the user still possesses the powerup.

4. The Hardware/Software Interface

4.1 Controller

We used a DragonRise Inc. Gamepad as an input source to interact with the game. There are eight buttons used in the game: ‘A’ starts the game, ‘Y’ and the left/right bumpers fire bullets, the left and right arrows control the horizontal movement of the ship, and the up and down arrows control the vertical movement.

The joypad is configured through libsub, and it follows the USB protocol.

VID	0x0079
PID	0x0011
Packet Size	8 bytes

Communication protocol

constant (0x01)	constant (0x7F)	constant (0x7F)	left/right arrow (0x7F)	up/down arrow (0x7F)	A/B/X/Y (0x0F)	triggers/ starts (0x00)	constant (0x00)

Controller Key assignment

A	starts the game	0x2F
Y	fires bullets	0xAF
Left bumper	fires bullets	0x01
Right bumper	fires bullets	0x02
Left & Right bumpers	fires bullets	0x03
Left arrow	move left	0x00
Right arrow	move right	0xFF
Up arrow	move up	0x00
Down arrow	move down	0xFF

5. Conclusion

5.1 Challenges and Lessons Learned

- Version control is essential. Keeping track of which hardware and software files communicate properly is important. Changing software and hardware at the same time introduces a lot of places for error, and it is very hard to figure out which piece is causing the issue.

- Talk to the professor early and often. Even if you are assigned a TA as your lead, the professor is almost always in the lab and happy to help. He is very responsive to emails and Ed posts, and can help with the most complex parts of your project.

- Come in with a strong design, lots of research, and careful thought about project implementation. If you come in with little plan, things will break quickly and your project will devolve into putting out fires without solving the main issues. This is another reason why it is so important to talk to the professor early, as he can help guide you through some of your tougher decisions.

- For hardware, at least from our experience, Quartus has been really hard to use. Some of its tools such as MegaWizard and generation of IPs are hard to use for reasons not limited to version deprecation and incompatibility. As an example, we spent multiple days trying to figure out adding custom width ROM using Quartus IPs, but was unsuccessful and chose to use Platform Designer directly instead to generate the IPs.

- Be very careful about using “for loops” and “division” in hardware code. It is very easy to unintentionally generate an excessive amount of logic. So for scores or any other hard calculation, we should better try to do calculation in software instead of hardware. Keep an eye on the generated flow reports on resource utilization and static timing analysis reports on “fmax” in output files after synthesis.

- Allow enough time for hardware work. The synthesis process takes some time which could be frustrating and the time of debugging could be unpredictable.

5.2 Contributions

Noah Hartzfeld , Zhengtao Hu: MIF file preprocessing; Joypad controller interface design; Game logic design, implementation, and testing; VGA h/s interface implementation, Presentation
Mingzhi Li, Jingyi Lai, Hiroki Endo: Tile and sprite generator design and implementation, .mif file preprocessing; ROM creation and module connection; color palette; Spirits load logic; VGA h/s interface implementation, Documentation

6. Code

6.1 Software

vga_ball.h

```C/C++
#ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>
#include <stdbool.h>

/* 定义最大子弹数量 */
#define SHIP_BULLETS 3
#define MAX_BULLETS 15
#define ENEMY_COUNT 60

#define SHIP_WIDTH 16
#define SHIP_HEIGHT 16
#define LIFE_COUNT 5

#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 480

#define SHIP 0 //
#define SHIP_LEFT 1 //
#define SHIP_RIGHT 2 //

#define SHIP_BULLET 9 //

#define ENEMY1 3 //

#define ENEMY2 5 //
```

```
#define ENEMY3 4 //

#define ENEMY_BULLET 10 //
#define ENEMY_BULLET_LEFT 11 //
#define ENEMY_BULLET_RIGHT 12 //

#define EXTRA_LIFE 6 //
#define SHIP_SPEED 7 //
#define EXTRA_BULLETS 8


#define SHIP_EXPLOSION1 13
#define SHIP_EXPLOSION2 14//
#define SHIP_FLAME 15


/* Color structure */
typedef struct {
   unsigned char red, green, blue;
} background_color;

typedef struct {
   unsigned short pos_x, pos_y;
   short velo_x, velo_y;
   short enemy;
   bool active;
} bullet;

typedef struct {
   unsigned short pos_x, pos_y;
   char sprite;
```

```
   bool active, indicator;
} powerup;

typedef struct {
   unsigned short pos_x, pos_y;
   short velo_x, velo_y;
   short lives, num_buls, explosion_timer, sprite;
   bullet bullets[SHIP_BULLETS];
   bool active;
} spaceship;

typedef struct {
   unsigned short pos_x, pos_y;
   short velo_x, velo_y;
   short start_x, start_y;

   short move_time, turn_counter, bul_cooldown, bul,
explosion_timer;
   char sprite, row, col, position;
   bool active, returning, moving;
} enemy;



typedef struct {
   spaceship ship;
   bullet bullets[MAX_BULLETS];
   enemy enemies[ENEMY_COUNT];
   background_color background;
   powerup power_up;
   int score;
} gamestate;
```

```
#define VGA_BALL_MAGIC 'v'

/* ioctls and their arguments */
#define UPDATE_ENEMIES   _IOW(VGA_BALL_MAGIC, 1, gamestate)
#define UPDATE_SHIP   _IOW(VGA_BALL_MAGIC, 2, spaceship)
#define UPDATE_SHIP_BULLETS   _IOW(VGA_BALL_MAGIC, 3, spaceship)
#define UPDATE_POWERUP   _IOW(VGA_BALL_MAGIC, 4, powerup)

#endif /* _VGA_BALL_H */
```

vga_ball.c

C/C++

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

#define DRIVER_NAME "vga_ball"
```

```
/* Device registers */
#define BG_COLOR(x)      (x)
#define OBJECT_DATA(x,i) ((x) + (4*(i)))

/*
* Information about our device
*/
struct vga_ball_dev {
   struct resource res; /* Resource: our registers */
   void __iomem *virtbase; /* Where registers can be accessed in
memory */
   background_color background;
   spaceship ship;
   bullet bullets[MAX_BULLETS];
   enemy enemies[ENEMY_COUNT];
   powerup power_up;
   int score;
} dev;

/*
* Write background color
*/
static void write_background(background_color *background)
{
   u32 color_data = ((u32)background->red << 16) |
                       ((u32)background->green << 8) |
                       background->blue;

   iowrite32(color_data, BG_COLOR(dev.virtbase));
   dev.background = *background;
}
```

```c
/*
* Write object data
*/
static void write_object(int idx, unsigned short x, unsigned short
y, char sprite_idx, char active)
{
   // 构建32位对象数据
   u32 obj_data = ((u32)(x & 0xFFF) << 20) |   // x位置 (12位)
               ((u32)(y & 0xFFF) << 8) |    // y位置 (12位)
               ((u32)(sprite_idx & 0x3F) << 2) | // 精灵索引 (6位)
               ((u32)(active & 0x1) << 1);  // 活动状态 (1位)

   iowrite32(obj_data, OBJECT_DATA(dev.virtbase, idx));
}

static void write_score(int idx, int score)
{
   u32 obj_data = (uint32_t)(score & 0xFFFFFFFF);

   iowrite32(obj_data, OBJECT_DATA(dev.virtbase, idx));

   dev.score = score;
}


static void write_ship(spaceship *ship){

   int sprite, i, active;

   if (ship->sprite == SHIP_EXPLOSION1) sprite = SHIP_EXPLOSION1;
```

```
   else if (ship->sprite == SHIP_EXPLOSION2) sprite =
SHIP_EXPLOSION2;

   else if (ship->velo_x < 0) sprite = SHIP_LEFT;

   else if (ship->velo_x > 0) sprite = SHIP_RIGHT;

   else sprite = SHIP;

   write_object (2, ship->pos_x,  ship->pos_y, sprite,
ship->active);

   if (ship->velo_y < 0 && ship->active & !ship->explosion_timer)
active = 1;
   else active = 0;

   write_object (3, ship->pos_x,  ship->pos_y+SHIP_HEIGHT,
SHIP_FLAME, active);

   dev.ship = *ship;

   for(i = 0; i<LIFE_COUNT; i++){

       if(i<ship->lives) active = 1;
       else active = 0;

       write_object (i+4, i*20+10,  SCREEN_HEIGHT-16, SHIP, active);
   }
}

static void write_ship_bullets(spaceship *ship){
```

```
   int i;
   bullet *bul;

   for (i = 0; i < SHIP_BULLETS; i++) {

       bul = &ship->bullets[i];
       write_object (i+LIFE_COUNT+4, bul->pos_x,  bul->pos_y,
SHIP_BULLET, bul->active);

       dev.ship.bullets[i] = *bul;
   }
}



/*
* Write all objects
*/
static void write_enemies(bullet bullets[], enemy enemies[])
{

   int i;
   bullet *bul;
   enemy *enemy;
   char sprite;

   for (i = 0; i < ENEMY_COUNT; i++) {

       enemy = &enemies[i];

       write_object(i+SHIP_BULLETS+LIFE_COUNT+4,  enemy->pos_x,
enemy->pos_y, enemy->sprite, enemy->active);
       dev.enemies[i] = enemies[i];
```

```
   }

   for (i = 0; i < MAX_BULLETS; i++) {

       bul = &bullets[i];

       if (bul->velo_x < 0) sprite = ENEMY_BULLET_LEFT;

       else if (bul->velo_x > 0) sprite = ENEMY_BULLET_RIGHT;

       else sprite = ENEMY_BULLET;

       write_object(i+SHIP_BULLETS+LIFE_COUNT+ENEMY_COUNT+4,
bul->pos_x,  bul->pos_y, sprite, bul->active);
       dev.bullets[i] = *bul;
   }
}



static void write_powerup(powerup *power_up){

   write_object (SHIP_BULLETS+LIFE_COUNT+ENEMY_COUNT+MAX_BULLETS+4,
power_up->pos_x,  power_up->pos_y, power_up->sprite,
power_up->active);

   dev.power_up = *power_up;

}


/*
```

```
* Update all game state at once
*/
static void update_enemies(gamestate *game_state)
{
   // write_background(&game_state->background);

   write_score(1, game_state->score);

   write_enemies(game_state->bullets, game_state->enemies);
}



static gamestate vb_arg;
static spaceship vb_ship;
static powerup vb_pu;

/*
* Handle ioctl() calls from userspace
*/
static long vga_ball_ioctl(struct file *f, unsigned int cmd,
unsigned long arg)
{
   switch (cmd) {
       case UPDATE_ENEMIES:
           if (copy_from_user(&vb_arg, (gamestate *) arg,
sizeof(gamestate)))
               return -EACCES;
           update_enemies(&vb_arg);
           break;

       case UPDATE_SHIP:
```

```
            if (copy_from_user(&vb_ship, (spaceship *) arg,
sizeof(spaceship)))
                return -EACCES;
            write_ship(&vb_ship);
            break;

        case UPDATE_SHIP_BULLETS:
            if (copy_from_user(&vb_ship, (spaceship *) arg,
sizeof(spaceship)))
                return -EACCES;
            write_ship_bullets(&vb_ship);
            break;

        case UPDATE_POWERUP:
            if (copy_from_user(&vb_pu, (powerup *) arg,
sizeof(powerup)))
                return -EACCES;
            write_powerup(&vb_pu);
            break;



        default:
            return -EINVAL;
    }

    return 0;
}

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

```
    .unlocked_ioctl = vga_ball_ioctl,
};

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

/*
* Initialization code: get resources and display initial state
*/
static int __init vga_ball_probe(struct platform_device *pdev)
{
    // Initial values
    background_color background = { 0x00, 0x00, 0x20 };

    int ret;

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

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

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
```

```
                        DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

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



    /* Set initial values */
    write_background(&background);

    return 0;

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

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

```
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
   { .compatible = "csee4840,vga_ball-1.0" },
   {},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

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

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

/* Called when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
   platform_driver_unregister(&vga_ball_driver);
```

```
   pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("VGA Ball Demo");
MODULE_DESCRIPTION("VGA Ball demo driver");
```

controller.h

C/C++

```
#ifndef _CONTROLLER_H
#define _CONTROLLER_H

#include <libusb-1.0/libusb.h>


typedef struct {
   uint8_t pad_1;
   uint8_t pad_2;
   uint8_t pad_3;
   uint8_t lr_arrows;
   uint8_t ud_arrows;
   uint8_t buttons;
   uint8_t bumpers; // maybe change name
   uint8_t pad_4;

} controller_packet;
```

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

#endif
```

controller.c

```C/C++
#include "controller.h"

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

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

/*
* Find and return a USB keyboard device or NULL if not found
* The argument con
*
```

```
*/
struct libusb_device_handle *opencontroller(uint8_t
*endpoint_address) {

 libusb_device **devs;
 struct libusb_device_handle *controller = NULL;
 struct libusb_device_descriptor desc;

 ssize_t num_devs, d;

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

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

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

 for (d = 0 ; d < num_devs ; d++) {

   libusb_device *dev = devs[d];

   if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
```

```
    fprintf(stderr, "Error: libusb_get_device_descriptor
failed\n");
    exit(1);
  }

  if (desc.idProduct == 0x0011) {
    struct libusb_config_descriptor *config;
    libusb_get_config_descriptor(dev, 0, &config);

    const struct libusb_interface_descriptor *inter =
    config->interface[0].altsetting;

    int r;

    if ((r = libusb_open(dev, &controller)) != 0) {
      fprintf(stderr, "Error: libusb_open failed: %s (%d)\n",
          libusb_error_name(r), r);
      exit(1);
    }

    if (libusb_kernel_driver_active(controller,0))
      libusb_detach_kernel_driver(controller, 0);

    libusb_set_auto_detach_kernel_driver(controller, 0);

    if ((r = libusb_claim_interface(controller, 0)) != 0) {
      fprintf(stderr, "Error: libusb_claim_interface failed: %d\n",
r);
      exit(1);
    }

    *endpoint_address = inter->endpoint[0].bEndpointAddress;
```

```
    goto found;
   }
 }

found:
 libusb_free_device_list(devs, 1);

 return controller;
}
```

hello.c (game logic)

C/C++

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include <pthread.h>
#include <fcntl.h>
#include "vga_ball.h"
#include "controller.h"
```

```
// #define SCREEN_WIDTH 1280


#define COLOR_COUNT 5

#define SHIP_INITIAL_X 300
#define SHIP_INITIAL_Y 400

#define BULLET_WIDTH 8
#define BULLET_HEIGHT 4


#define ENEMY_WIDTH 16
#define ENEMY_HEIGHT 16

#define ENEMY_SPACE 10
#define COLUMNS 22

#define ENEMY3_BULLET_COOLDOWN 50

#define ENEMY4_BULLET_COOLDOWN 40


#define LEFT_ARROW 0x00
#define RIGHT_ARROW 0xff
#define UP_ARROW 0x00
#define DOWN_ARROW 0xff
#define BUTTON_A 0x2f
#define Y_BUTTON 0x8f
#define LEFT_BUMPER 0x01
#define RIGHT_BUMPER 0x02
#define LR_BUMPER 0x03
```

```
#define NO_INPUT 0x7f // ????????????????



/* File descriptor for the VGA ball device */
static int vga_ball_fd;


#define NUM_ROWS 5
static char row_vals[NUM_ROWS] = {0,0,0,0,1};
// static char row_vals[NUM_ROWS] = {0,4,3,2,1};
// static char row_vals[NUM_ROWS] = { 2, 6, 8, 10, 10 };
static char row_sprites[NUM_ROWS] = { ENEMY1, ENEMY2,ENEMY2, ENEMY3,
ENEMY3};
static int row_fronts[NUM_ROWS];
static int row_backs[NUM_ROWS];


static int kill_count = 0;

static int ship_velo = 2;

static int powerup_timer = 0;
#define EXTRA_BULLET_TIME 300;
#define EXTRA_SPEED_TIME 750;


#define EXPLOSION_TIME 10


static int blink_counter = 0;
```

```
#define BLINK_COUNT 10
#define QUICK_BLINK_COUNT 5


static int round_frequency = 100;



static int enemy_wiggle = 1;
static int enemy_wiggle_time = 0;


static int moving = 300;

static const char filename[] = "/dev/vga_ball";

/* Array of background colors to cycle through */
static const background_color colors[] = {
   { 0x00, 0x00, 0x10 },  // Very dark blue
   { 0x00, 0x00, 0x20 },  // Dark blue
   { 0x10, 0x10, 0x30 },  // Navy blue
   { 0x00, 0x00, 0x40 },  // Medium blue
   { 0x20, 0x20, 0x40 }   // Blue-purple
};



#define TURN_TIME 70
static short turn_x[TURN_TIME] = {2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,
                  2,2,2,2,2,2,2,2,
```

```
                    2,2,2,2,2,2,2,2,
                    2,2,2,2,2,2,2,2,
                    2,2,2,2,2,2,2,2,
                    0,0,0,0,0,0,0,0,
                    0,0,0,0,0,0,0,0,
                    0,0,0,0,0,0};

static short turn_y[TURN_TIME] = {-2,-2,-2,-2,-2,-2,-2,-2,
                    -2,-2,-2,-2,-2,-2,-2,-2,
                    -2,-2,-2,-2,-2,-2,-2,-2,
                    0,0,0,0,0,0,0,0,
                    2,2,2,2,2,2,2,2,
                    2,2,2,2,2,2,2,2,
                    2,2,2,2,2,2,2,2,
                    2,2,2,2,2,2,2,2,
                    2,2,2,2,2,2};



static int num_enemies_moving = 0, active_ship_buls = 0,
active_enemy_buls = 0;



static int round_wait_time = 0;
static long round_time = 0;
static int active1 = 0, active2 = 0, active3 = 0, round_pause,
num_sent, send_per_round = 20;
#define TOTAL_ACTIVE (active1 + active2 + active3)
#define ROUND_WAIT 100
static int round_num = 1;
```

```
static gamestate game_state = {

   .ship = {.pos_x = SHIP_INITIAL_X, .pos_y = SHIP_INITIAL_Y,
.velo_x = 0, .velo_y = 0, .lives = LIFE_COUNT, .num_buls = 3,
.bullets = { 0 }, .active = 1},
   .background = {.red = 0x00, .green = 0x00, .blue = 0x20},
   .bullets = { 0 },
   .enemies = { 0 },
   .power_up = { 0 },
   .score = 0
};

/**
* Update game state and send to the device
*/
void update_enemies() {
   if (ioctl(vga_ball_fd, UPDATE_ENEMIES, &game_state)) {
       perror("ioctl(UPDATE_ENEMIES) failed");
       exit(EXIT_FAILURE);
   }
}


void update_ship() {
   if (ioctl(vga_ball_fd, UPDATE_SHIP, &game_state.ship)) {
       perror("ioctl(UPDATE_SHIP) failed");
       exit(EXIT_FAILURE);
   }
}

void update_ship_bullet() {
   if (ioctl(vga_ball_fd, UPDATE_SHIP_BULLETS, &game_state.ship)) {
```

```
        perror("ioctl(UPDATE_SHIP_BULLETS) failed");
        exit(EXIT_FAILURE);
    }
}


void update_powerup() {
    if (ioctl(vga_ball_fd, UPDATE_POWERUP, &game_state.power_up)) {
        perror("ioctl(UPDATE_POWERUP) failed");
        exit(EXIT_FAILURE);
    }
}



void apply_powerup(powerup *power_up){

    spaceship *ship = &game_state.ship;

    switch (power_up->sprite){

        case EXTRA_LIFE:

            ship->lives++;

            // draw an extra ship life
            break;

        case SHIP_SPEED:

            ship_velo = 3;
            powerup_timer = EXTRA_SPEED_TIME;
```

```
            break;

        case EXTRA_BULLETS:

            ship->num_buls = 3;
            powerup_timer = EXTRA_BULLET_TIME;
            break;
    }
}

void active_powerup(){

    powerup *power_up = &game_state.power_up;

    if (--powerup_timer < 0){

        game_state.ship.num_buls = 5;
        ship_velo = 2;

    }

    else if (powerup_timer == 0){
        power_up->active = 0;
        blink_counter = 0;
    }
    else if(powerup_timer > 50 && powerup_timer < 100){

        if (blink_counter == 0){

            blink_counter = BLINK_COUNT;
            power_up->active = !power_up->active;
```

```
        }
        else{
            blink_counter --;
        }
    }
    else if (powerup_timer > 0 && powerup_timer < 50){

        if (blink_counter == 0){

            blink_counter = QUICK_BLINK_COUNT;
            power_up->active = !power_up->active;

        }
        else{
            blink_counter --;
        }
    }
}

void move_powerup(){

    powerup *power_up = &game_state.power_up;
    spaceship *ship = &game_state.ship;

    if (power_up->active && !power_up->indicator){

        power_up->pos_y += 1;

        if (ship->active &&
            abs(ship->pos_x - power_up->pos_x) <= SHIP_WIDTH &&
            abs(ship->pos_y - power_up->pos_y) <= SHIP_HEIGHT){
```

```
            apply_powerup(power_up);

            if(power_up->sprite == EXTRA_LIFE)
                power_up->active = 0;

            power_up->pos_x = 400;
            power_up->pos_y = SCREEN_HEIGHT-SHIP_HEIGHT;
            power_up->indicator = 1;


            kill_count = 0;
        }

        if (power_up->pos_y >= SCREEN_HEIGHT){

            power_up->active=0;
            kill_count = 10;
        }

    }

}

void drop_powerup(enemy *enemy){

    powerup *power_up = &game_state.power_up;
    int i = rand() % 3;

    if (game_state.ship.lives == LIFE_COUNT && i == 2)
        while (i == 2) i = rand() % 3;

    power_up->pos_x = enemy->pos_x;
```

```
    power_up->indicator = 0;

    power_up->pos_y = 200;
    power_up->active = 1;

    switch (i){
        case 0:
            power_up->sprite = SHIP_SPEED;

            break;

        case 1:
            power_up->sprite = EXTRA_BULLETS;
            break;

        case 2:
            power_up->sprite = EXTRA_LIFE;
            break;

        default:
            break;
    }
}



void change_active_amount(char enemy_sprite){

    switch(enemy_sprite){

        case ENEMY1:
```

```
            active1 --;
            break;

        case ENEMY2:
            active2 --;
            break;

        case ENEMY3:
            active3 --;
            break;
    }
}


void change_score(char sprite){

    switch(sprite){

        case ENEMY1:
            game_state.score += 20;
            break;

        case ENEMY2:
            game_state.score += 10;
            break;

        case ENEMY3:
            game_state.score += 5;
            break;

        case SHIP:
            game_state.score -= 50;
```

```
    }


    if(game_state.score < 0) game_state.score = 0;

}


void calculate_velo(int ship_x, int ship_y, void *object, int type,
short scaler){

    float new_x, new_y, mag;
    enemy *enemyy;
    bullet *bul;

    if (type) {

        enemyy = (enemy *)object;

        new_x = ship_x - enemyy->pos_x;
        new_y = ship_y - enemyy->pos_y;
    }

    else {

        bul = (bullet *)object;

        new_x = ship_x - bul->pos_x;
        new_y = ship_y - bul->pos_y;
    }

    mag = sqrt(new_x * new_x + new_y * new_y);
```

```
   new_x /= mag;
   new_y /= mag;

   new_x *= scaler;
   new_y *= scaler;

   if (type) {

       enemyy->velo_x = (short)new_x;
       enemyy->velo_y = (short)new_y;
   }
   else{

       bul->velo_x = (short)new_x;
       bul->velo_y = (short)new_y;
   }
}


void change_row_ends(int cur_end, int row_num, int front){


   if(!front){

       for (int i=cur_end-1; game_state.enemies[i].row == row_num;
i--){

           if(game_state.enemies[i].active &&
!game_state.enemies[i].moving){

               row_backs[row_num] = i;
```

```
                break;
            }
        }
    }
    else{

        for (int i=cur_end+1; game_state.enemies[i].row == row_num;
i++){

            if(game_state.enemies[i].active &&
!game_state.enemies[i].moving){

                row_fronts[row_num] = i;
                break;
            }
        }
    }
}



bool aquire_bullet(enemy *enemy){

    bullet *bul;

    for (int i = 0; i<MAX_BULLETS; i++){

        bul = &game_state.bullets[i];

        if(!bul->active && game_state.ship.active){

            bul->active = 1;
```

```
            bul->pos_x = enemy->pos_x;
            bul->pos_y = enemy->pos_y+(ENEMY_HEIGHT);

            bul->enemy = enemy->position;

            enemy->bul = i;

            active_enemy_buls ++;

            return 1;
        }
    }

    return 0;


}


void enemy_shoot(enemy *enemy){

    spaceship *ship = &game_state.ship;

    bool aquired;

    if (ship->active && enemy->bul_cooldown <= 0 &&
        enemy->turn_counter >= TURN_TIME && !enemy->returning){

        if (enemy->sprite == ENEMY2){

            if (abs(ship->pos_x - enemy->pos_x) <= 80
```

```
                    && abs(ship->pos_y - enemy->pos_y) <= 150
                    && ship->pos_y - 30 > enemy->pos_y){

                if (enemy->bul == -1){

                    if((aquired = aquire_bullet(enemy))){

                        enemy->bul_cooldown = ENEMY3_BULLET_COOLDOWN;
                        game_state.bullets[enemy->bul].velo_y = 3;
                        game_state.bullets[enemy->bul].velo_x = 0;

                    }
                }
            }
        }

        else if(enemy->sprite == ENEMY3){

            if (abs(ship->pos_x - enemy->pos_x) <= 150
                && abs(ship->pos_y - enemy->pos_y <= 200
                && ship->pos_y - 60 > enemy->pos_y)){

                if (enemy->bul == -1){

                    if((aquired = aquire_bullet(enemy))){

                        enemy->bul_cooldown = ENEMY4_BULLET_COOLDOWN;
                        calculate_velo(ship->pos_x, ship->pos_y,
&game_state.bullets[enemy->bul], 0, 4);

                    }
                }
```

```
            }
        }
    }

    else if(enemy->turn_counter <= TURN_TIME)
        enemy->bul_cooldown --;

}


void enemy_return (enemy *enemy){

    int position;


    if (enemy->pos_y > SCREEN_HEIGHT || enemy->pos_x > SCREEN_WIDTH
|| enemy->pos_x < 0){

        enemy->returning = 1;

        enemy->pos_x = enemy->start_x;
        enemy->pos_y = 0;

        calculate_velo(enemy->start_x + enemy_wiggle_time,
enemy->start_y, enemy, 1, 2);
    }


    if (enemy->returning){

        if (abs(enemy->pos_x - enemy->start_x + enemy_wiggle_time) <
25 && abs(enemy->pos_y -enemy->start_y) < 25){
```

```
            enemy->pos_x = enemy->start_x+enemy_wiggle_time;
            enemy->pos_y = enemy->start_y;

            enemy->velo_x = 0;
            enemy->velo_y = 0;

            enemy->moving = 0;
            enemy->returning = 0;
            enemy->move_time = 0;
            enemy->turn_counter = 0;

            num_enemies_moving --;

        }

        else
            calculate_velo(enemy->start_x + enemy_wiggle_time,
enemy->start_y, enemy, 1, 2);

    }

}


void turn(enemy *enemy){

    spaceship *ship = &game_state.ship;


    if (enemy->start_x <= SCREEN_WIDTH/2)
            enemy->velo_x = -turn_x[enemy->turn_counter];
```

```
  else
      enemy->velo_x = turn_x[enemy->turn_counter];


  enemy->velo_y = turn_y[enemy->turn_counter];
  enemy->turn_counter++;


  if (enemy->turn_counter == TURN_TIME){


      if(enemy->sprite == ENEMY1){

          enemy->velo_x = (enemy->pos_x < SCREEN_WIDTH / 2) ? 2 :
-2;
          enemy->velo_y = 1;
      }

      else if(enemy->sprite == ENEMY2){

          enemy->velo_x = (enemy->pos_x < SCREEN_WIDTH / 2) ? 4 :
-4;
          enemy->velo_y = 2;
      }
      else {

          calculate_velo(ship->pos_x, ship->pos_y, enemy, 1, 3);
      }

  }
```

```
}


void enemy_attack(enemy *enemy){


   spaceship *ship = &game_state.ship;
   int cont;


   enemy->pos_x += enemy->velo_x;
   enemy->pos_y += enemy->velo_y;


   if (enemy->turn_counter < TURN_TIME)
       turn(enemy);

   else{

       if (enemy->sprite == ENEMY1){

           if (!ship->active){

               enemy->velo_x = 0;
               enemy->velo_y = 4;
           }

           else if(++enemy->move_time < 250)
               calculate_velo(ship->pos_x, ship->pos_y, enemy, 1,
3);
           else{
```

```
                enemy->velo_x = 0;
                enemy->velo_y = 2;
            }
        }

        else if (enemy->sprite == ENEMY2){


            if (enemy->pos_y+30 >= ship->pos_y || !ship ->active) {

                enemy->velo_x = (enemy->pos_x > ship->pos_x) ? 1 :
-1;
                enemy->velo_y = 4;

            }

            else if (enemy->move_time == 0){

                if (enemy->start_x < SCREEN_WIDTH/2 && enemy->pos_x -
ship->pos_x > 10 && ship->pos_y - enemy->pos_y < 150){

                    calculate_velo(ship->pos_x, ship->pos_y, enemy,
1, 2);
                    enemy->move_time++;
                }

                else if (ship->pos_x - enemy->pos_x > 10 &&
ship->pos_y - enemy->pos_y < 150){

                    calculate_velo(ship->pos_x, ship->pos_y, enemy,
1, 2);
                    enemy->move_time++;
```

```
            }
        }

        else{

            if(enemy->move_time < 75){

                if (enemy->start_x < SCREEN_WIDTH/2)
                    // calculate_velo(ship->pos_x -200,
ship->pos_y, enemy, 1, 2);
                    enemy->velo_x = -2;

                else
                    // calculate_velo(ship->pos_x +200,
ship->pos_y, enemy, 1, 2);
                    enemy->velo_x = 2;
            }

            else{

                if (enemy->start_x < SCREEN_WIDTH/2)
                    // calculate_velo(ship->pos_x +200,
ship->pos_y, enemy, 1, 2);
                    enemy->velo_x = 2;
                else
                    // calculate_velo(ship->pos_x -200,
ship->pos_y, enemy, 1, 2);
                enemy->velo_x = -2;
            }

            if(++ enemy->move_time > 150)
```

```
                    calculate_velo(ship->pos_x, ship->pos_y, enemy,
1, 2);
            }
        }

        else{

            if (!ship->active){

                enemy->velo_x = (enemy->pos_x > ship->pos_x) ? 1 :
-1;
                enemy->velo_y = 4;
            }

            else if(++enemy->move_time == 150){

                cont = rand() % 4;

                if(!cont)
                    enemy->velo_x = -enemy->velo_x;

                else
                    enemy->move_time --;
            }

            else if(enemy->move_time == 250){

                enemy->velo_x = 0;
                enemy->velo_y = 2;
            }
            else if (enemy->pos_y > ship->pos_y){
```

```
                enemy->velo_x = (enemy->pos_x > ship->pos_x) ? 1 :
-1;
                enemy->velo_y = 2;
            }
        }

    }

    enemy_return(enemy);

}


int enemy_explosion(){

    enemy *enemy;
    int num_left = 0;

    for(int i = 0; i<ENEMY_COUNT; i++){

        enemy = &game_state.enemies[i];

        if(enemy->explosion_timer == 1){
                        printf("33333333333333333\n");

            memset(enemy, 0, sizeof(*enemy));
        }

        else if(enemy->explosion_timer < EXPLOSION_TIME/2 &&
enemy->explosion_timer){
                        printf("22222222222\n");
```

```
            enemy->sprite = SHIP_EXPLOSION2;
            enemy->explosion_timer --;
            num_left ++;
        }

        else if (enemy->explosion_timer){

            printf("1111111111\n");

            num_left ++;

            enemy->velo_x = 0;
            enemy->velo_y = 0;
            enemy->sprite = SHIP_EXPLOSION1;

            enemy->explosion_timer --;

        }
    }

    return num_left;
}


void ship_explosion(){

    spaceship *ship = &game_state.ship;

    if(ship->explosion_timer == 1){
        ship->active = 0;
        ship->explosion_timer = 0;
        ship->sprite = SHIP;
```

```
        ship->lives --;

    }
    else if(ship->explosion_timer < EXPLOSION_TIME/2 &&
ship->explosion_timer){
        ship->sprite = SHIP_EXPLOSION2;
        ship->explosion_timer --;
    }
    else if (ship->explosion_timer){


        ship->sprite = SHIP_EXPLOSION1;
        ship->explosion_timer --;

    }
}



int enemy_movement(int rand_enemy){

    int cont, row_num, num_left = 0;
    enemy *enemy;
    spaceship *ship = &game_state.ship;

    if (rand_enemy != -1){

        switch(rand_enemy){

            case ENEMY1:
                row_num = 0;
                break;
```

```
            case ENEMY2:
                row_num = 1 + rand() % 2;
                break;

            case ENEMY3:
                row_num = 3 + rand() % 2;
                break;
        }

        if (enemy_wiggle > 0) rand_enemy = row_fronts[row_num];
        else rand_enemy = row_backs[row_num];

    }

    for (int i = 0; i < ENEMY_COUNT; i++){

        enemy = &game_state.enemies[i];

        if (enemy->active && !enemy->explosion_timer){

            num_left++;

            if(!enemy->moving && rand_enemy == i){

                if (enemy_wiggle > 0) change_row_ends(i, row_num, 1);

                else change_row_ends(i, row_num, 0);

                enemy->velo_x = (enemy->start_x < SCREEN_WIDTH/2) ?
-1 : 1;
                enemy->velo_y = -4;
```

```
                enemy->moving = 1;
                num_enemies_moving ++;
            }

            if(enemy->moving) {

                enemy_attack(enemy);

                if (!enemy->moving){

                    if(i > row_backs[enemy->row] ||
!game_state.enemies[row_backs[enemy->row]].active)

                        row_backs[enemy->row] = i;


                    if(i < row_fronts[enemy->row] ||
!game_state.enemies[row_fronts[enemy->row]].active)

                        row_fronts[enemy->row] = i;
                }

                else
                    enemy_shoot(enemy);
            }

            else
                enemy->pos_x += enemy_wiggle;
```

```
            if (ship->active && !ship->explosion_timer &&
                abs(ship->pos_x - enemy->pos_x) <= SHIP_WIDTH
                && abs(ship->pos_y - enemy->pos_y) <= SHIP_HEIGHT){

                enemy->active = 0;

                if(i == row_backs[enemy->row]) change_row_ends(i,
enemy->row, 0);

                else if (i == row_fronts[enemy->row])
change_row_ends(i, enemy->row, 1);

                change_active_amount(enemy->sprite);

                if(enemy->moving) num_enemies_moving --;

                memset(enemy, 0, sizeof(*enemy));

                change_score(SHIP);

                ship->explosion_timer = EXPLOSION_TIME;
                enemy->explosion_timer = EXPLOSION_TIME;

                round_wait_time = ROUND_WAIT;
                num_left --;


            }
        }
    }
    return num_left;
```

```
}

void move_enemy_bul(){

   spaceship *ship = &game_state.ship;
   bullet *bul;

   for (int i = 0; i<MAX_BULLETS; i++){

       bul = &game_state.bullets[i];

       if(!bul->active) continue;

       bul->pos_x += bul->velo_x;
       bul->pos_y += bul->velo_y;


       if (ship->active && !ship->explosion_timer &&
           abs(ship->pos_x - bul->pos_x ) <= SHIP_WIDTH &&
           abs(ship->pos_y - bul->pos_y ) <= SHIP_HEIGHT){


           game_state.enemies[bul->enemy].bul = -1;

           bul->active = 0;
           bul->enemy = -1;

           active_enemy_buls --;

           change_score(SHIP);

           ship->lives --;
```

```
            ship->explosion_timer = EXPLOSION_TIME;

            round_wait_time = ROUND_WAIT;

        }

        if (bul->pos_y >= SCREEN_HEIGHT || bul->pos_x >= SCREEN_WIDTH
|| bul->pos_x < 0){

            game_state.enemies[bul->enemy].bul = -1;

            bul->active = 0;
            bul->enemy = -1;

            active_enemy_buls --;
        }
    }
}



void bullet_colision(bullet *bul){

    enemy *enemy;

    for (int i = 0; i<ENEMY_COUNT; i++){

        enemy = &game_state.enemies[i];

        if (enemy->explosion_timer) continue;

        if (enemy->active &&
```

```
            abs(enemy->pos_x - bul->pos_x) <= ENEMY_WIDTH &&
            abs(enemy->pos_y - bul->pos_y) <= ENEMY_HEIGHT){

            if(i == row_backs[enemy->row]) change_row_ends(i,
enemy->row, 0);

            else if (i == row_fronts[enemy->row]) change_row_ends(i,
enemy->row, 1);

            change_active_amount(enemy->sprite);

            bul->active = 0;

            active_ship_buls --;

            if (++ kill_count >= 15 && !game_state.power_up.active &&
                game_state.ship.active &&
!game_state.ship.explosion_timer)
                    drop_powerup(enemy);

            change_score(enemy->sprite);

            if(enemy->moving) num_enemies_moving --;

            enemy->explosion_timer = EXPLOSION_TIME;

            break;
        }
    }
}

void bullet_movement(int new_bullet){
```

```
   bullet *bul;
   int num_active = 0;

   for(int i = 0; i< SHIP_BULLETS; i++)
       if(game_state.ship.bullets[i].active) num_active++;

   for (int i = 0; i < SHIP_BULLETS; i++) {

       bul = &game_state.ship.bullets[i];

       if (bul->active){

           bul->pos_y += bul->velo_y;

           if (bul->pos_y <= 5){

               bul->active = 0;
               active_ship_buls --;
               continue;
           }

           bullet_colision(bul);
       }

       else if (!bul->active && new_bullet && num_active <
game_state.ship.num_buls) {
           bul->active = 1;
           bul->pos_x = game_state.ship.pos_x;
           bul->pos_y = game_state.ship.pos_y-(SHIP_HEIGHT);
           bul->velo_y = -3;
           new_bullet = 0;
```

```
            active_ship_buls ++;
        }
    }
}



void ship_movement(){

    spaceship *ship = &game_state.ship;

    if(ship->velo_x > 0 && ship->pos_x < SCREEN_WIDTH-SHIP_WIDTH-5)
        ship->pos_x += ship->velo_x;

    else if(ship->velo_x < 0 && ship->pos_x > 5)
        ship->pos_x += ship->velo_x;


    if (ship->velo_y > 0 && ship->pos_y <
SCREEN_HEIGHT-SHIP_HEIGHT*2-5)
        ship->pos_y += ship->velo_y;

    else if (ship->velo_y < 0 && ship->pos_y > 5)
        ship->pos_y += ship->velo_y;

}

// taking too long to move
// after so long I can have liek 5 go at the same time just remove %
int enemies_to_move(){
```

```
   enemy *enemy;
   int rand_enemy;


   if (TOTAL_ACTIVE != 0){

       rand_enemy = rand() % TOTAL_ACTIVE;

       if (rand_enemy < active1)
           rand_enemy = ENEMY1;

       else if (rand_enemy < active1 + active2)
           rand_enemy =  ENEMY2;

       else
           rand_enemy = ENEMY3;

       if (num_sent == send_per_round){

           if (!num_enemies_moving){

               num_sent = 0;
               round_pause = ROUND_WAIT/2;
           }
       }
       else if (num_sent > send_per_round/4 && num_sent <=
send_per_round/4 +3){

           num_sent ++;
           return rand_enemy;
       }
```

```
       else if (num_sent > send_per_round*3/4 && num_sent <=
send_per_round*3/4 +3){

           num_sent ++;
           return rand_enemy;
       }
       else{

           if(round_time % round_frequency == 0) {

               printf("%ld \n", round_time);

               num_sent ++;
               return rand_enemy;
           }

           else return -1;
       }
   }

   return -1;

}

void init_round_state() {

   int space, row = 0, enemy_count;

   enemy *enemy;

   enemy_count = row_vals[row];
```

```
    space = COLUMNS - row_vals[row];

    row_fronts[row] = 0;

    for (int i = 0, j=0; i < ENEMY_COUNT; i++, j++) {

        enemy = &game_state.enemies[i];

        memset(enemy, 0, sizeof(*enemy));

        while (i >= enemy_count && row < 5){

            row_backs[row] = i-1;

            row++;

            j = 0;

            space = COLUMNS - row_vals[row];
            enemy_count += row_vals[row];

            row_fronts[row] = i;
        }

        if (row < 5){

            enemy->pos_x = enemy->start_x = 50 + ((ENEMY_WIDTH +
ENEMY_SPACE) * (space / 2)) \
                + j * (ENEMY_WIDTH + ENEMY_SPACE);

            enemy->pos_y = enemy->start_y = 60 + 30 *(row+1);
            enemy->sprite = row_sprites[row];
```

```
            enemy->position = i;
            // enemy->active = 1;
            enemy->bul = -1;
            enemy->row = row;
            enemy->col = (space/2) + j;

            switch(row_sprites[row]){

                case ENEMY1:
                    active1 ++;
                    break;

                case ENEMY2:
                    active2 ++;
                    break;

                case ENEMY3:
                    active3 ++;
                    break;
            }
        }
        else
            enemy->col = -1;
    }
}




// static int x_coords[40] = {
//     // Y
//     50, 66, 82, 66, 66,
```

```
//     // O
//     134, 150, 166, 134, 166, 134, 150, 166,
//     // U
//     182, 198, 214, 182, 214, 182, 198, 214,
//     // W
//     246, 262, 278, 294, 262,
//     // I
//     310, 326, 342, 326, 326,
//     // N
//     358, 358, 374, 390, 390,
//     // !
//     406, 406, 406, 406
// };

// static int y_coords[40] = {
//     // Y
//     50, 50, 50, 66, 82,
//     // O
//     50, 50, 50, 66, 66, 82, 82, 82,
//     // U
//     50, 50, 50, 66, 66, 82, 82, 82,
//     // W
//     50, 50, 50, 50, 66,
//     // I
//     50, 50, 50, 66, 82,
//     // N
//     50, 66, 66, 82, 50,
//     // !
//     50, 66, 82, 114
// };
```

```c
struct libusb_device_handle *controller;

uint8_t endpoint_address;


int main(){

   spaceship *ship = &game_state.ship;
   controller_packet packet;
   int transferred, start = 0, new_bullet, prev_bullet = 0,
enemies_remaining, enemies_exploding, rand_enemy, save_score;
   int col_active = 0, active_buls = 0, active_enemies = 0;
   int bumpers = 0, buttons = 0;

   srand(time(NULL));


   /* Open the device file */
   if ((vga_ball_fd = open(filename, O_RDWR)) == -1) {
       fprintf(stderr, "Could not open %s\n", filename);
       return EXIT_FAILURE;
   }

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

```
   printf("Press A \n");

   while (start == 0){
       // recieve packets
       libusb_interrupt_transfer(controller, endpoint_address,
           (unsigned char *) &packet, sizeof(packet), &transferred,
0);

       if(packet.buttons == BUTTON_A) start = 1;
   }

   printf("Game Begins! \n");

   init_round_state();
   update_ship();

   usleep(16000);



   // for (int i = 0; i<40; i++){

   //     game_state.enemies[i]->sprite = SHIP_BULLET;
   //     game_state.enemies[i]->pos_x = x_coords[i];
   //     game_state.enemies[i]->pos_x = x_coords[i];
   //     game_state.enemies[i].active = 1;
   // }

   // update_enemies();
```

```
   for (int i =0; i<COLUMNS; i++){
       for(int j=0; j<ENEMY_COUNT; j++)
           if(game_state.enemies[j].col == i)
game_state.enemies[j].active = 1;

       update_enemies();
       usleep(16000);
   }

   for (;;){

       round_time++;
       active_buls = 0;
       active_enemies = 0;

       enemy_wiggle_time += enemy_wiggle;
       if (abs(enemy_wiggle_time) == 80) enemy_wiggle =
-enemy_wiggle;

       new_bullet = 0;

       if (ship->lives == 0) break;

       libusb_interrupt_transfer(controller, endpoint_address,
           (unsigned char *) &packet, sizeof(packet), &transferred,
0);

       if (transferred == sizeof(packet)) {

           switch (packet.lr_arrows) {
               case LEFT_ARROW:
                   if(ship->pos_x > 0)
```

```
                    ship->velo_x = -ship_velo;

                // printf("%d, %d \n", ship->pos_x, ship->pos_y);
                break;

            case RIGHT_ARROW:
                if(ship->pos_x < SCREEN_WIDTH-SHIP_WIDTH)
                    ship->velo_x = ship_velo;

                // printf("%d, %d \n", ship->pos_x, ship->pos_y);
                break;

            default:
                ship->velo_x = 0;
                break;
        }

        switch (packet.ud_arrows) {
            case UP_ARROW:
                if (ship->pos_y < SCREEN_HEIGHT - 5)
                    ship->velo_y = -ship_velo;

                // printf("%d, %d\n", ship->pos_x, ship->pos_y);
                break;

            case DOWN_ARROW:
                if (ship->pos_y > 0+SHIP_HEIGHT)
                    ship->velo_y = ship_velo;

                // printf("%d, %d \n", ship->pos_x, ship->pos_y);
                break;
```

```
                default:
                    ship->velo_y = 0;
                    break;
            }

            switch (packet.buttons) {
                case Y_BUTTON:
                    if (!prev_bullet ){
                        new_bullet = 1; // do not allow them to hold
the button to shoot
                        prev_bullet = 1;
                    }

                    buttons = 1;
                    // printf("Bullet \n");
                    break;

                default:
                    if (!bumpers) prev_bullet = 0;
                    buttons = 0;
                    break;
            }

            switch (packet.bumpers) {
                case LEFT_BUMPER:
                    if (!prev_bullet){
                        new_bullet = 1; // do not allow them to hold
the button to shoot
                        prev_bullet = 1;
                    }

                    bumpers = 1;
```

```
                break;

            case RIGHT_BUMPER:
                if (!prev_bullet){
                    new_bullet = 1; // do not allow them to hold
the button to shoot
                    prev_bullet = 1;
                }

                bumpers = 1;

                break;

            case LR_BUMPER:
                if (!prev_bullet ){
                    new_bullet = 1; // do not allow them to hold
the button to shoot
                    prev_bullet = 1;
                }
                bumpers = 1;
                break;

            default:
                if (!buttons) prev_bullet = 0; // only reset
bullets if the y button has not been pressed
                bumpers = 0;
                // printf("bumpers\n");
                break;
        }
```

```
            if(ship->active && !ship->explosion_timer)
ship_movement();

            move_powerup();
            enemies_exploding = enemy_explosion();
            ship_explosion();

            if (!round_wait_time){ // ship is alive and round is
playing

                active_powerup();

                if(ship->active) bullet_movement(new_bullet);

                rand_enemy = enemies_to_move();
                enemies_remaining = enemy_movement(rand_enemy);
                move_enemy_bul();

            }

            else if(round_wait_time == 1){

                if(!ship->active){

                    ship->active = 1;
                    ship->pos_x = SHIP_INITIAL_X;
                    ship->pos_y = SHIP_INITIAL_Y;
                    round_wait_time = 0;
                    round_time = 0;

                    num_sent = 0;
```

```
                    powerup_timer = 0;
                    kill_count /= 2;

                }

                else{

                    for(int j=0; j<ENEMY_COUNT; j++)
                        if(game_state.enemies[j].col == col_active)
game_state.enemies[j].active = 1;

                    if (++col_active == COLUMNS) round_wait_time = 0;
                }
            }

            else{

                game_state.power_up.active = 0;

                // printf("%d, %d, %d \n", active_ship_buls,
active_enemy_buls, num_enemies_moving);

                if(!active_ship_buls && !active_enemy_buls &&
!num_enemies_moving)
                    round_wait_time --;

                if (round_wait_time > 30) round_wait_time --;

                enemy_movement(-1);
                move_enemy_bul();
                bullet_movement(0);
```

```
        }

        update_ship();
        update_enemies();
        update_powerup();
        update_ship_bullet();

        if(ship->lives <= 0){
            printf("You lost =( \n");

            save_score = game_state.score;

            memset(&game_state, 0, sizeof(gamestate));

            game_state.score = save_score;

            update_ship();
            update_enemies();
            update_powerup();
            update_ship_bullet();


            break;
        }

        if(!enemies_remaining && !ship->explosion_timer &&
            !round_wait_time && !enemies_exploding){

            if(round_num == 3){

                printf("You Won!");
```

```
            save_score = game_state.score;

            memset(&game_state, 0, sizeof(gamestate));

            game_state.score = save_score;

            update_ship();
            update_enemies();
            update_powerup();
            update_ship_bullet();

            break;
        }

        if(!active_enemy_buls){

            enemy_wiggle_time = 0;
            enemy_wiggle = 1;

            round_wait_time = ROUND_WAIT;
            col_active = 0;

            round_time = 0;
            num_sent = 0;

            round_frequency -=25;

            send_per_round += send_per_round/4;

            active1 = active2 = active3 = 0;

            row_vals[0] ++;
```

```
                    for(int i =1; i<5; i++){

                        row_vals[i] += round_num*2;
                    }

                    init_round_state();

                    enemies_remaining = 1;
                    round_num++;

                }

            }

            usleep(16000);
        }
    }

}
```

6.2 Hardware

vga_ball.sv

```
Unset
module vga_ball#(
   parameter MAX_OBJECTS = 100,
   parameter SPRITE_WIDTH = 16,
   parameter SPRITE_HEIGHT = 16,
   parameter TILE_WIDTH   = 16,
   parameter TILE_HEIGHT  = 16
) (
   input  logic        clk,
   input  logic        reset,
   input  logic [31:0] writedata,
   input  logic        write,
   input  logic        chipselect,
   input  logic [6:0]  address,
   output logic [7:0]  VGA_R, VGA_G, VGA_B,
   output logic        VGA_CLK, VGA_HS, VGA_VS,
   output logic        VGA_BLANK_n,
   output logic        VGA_SYNC_n
);

   logic [10:0]    hcount;
   logic [9:0]     vcount;

   // Background color
   logic [7:0]     background_r, background_g, background_b;
   logic [7:0]     score;

   logic [11:0]    obj_x[MAX_OBJECTS];
   logic [11:0]    obj_y[MAX_OBJECTS];
   logic [5:0]     obj_sprite[MAX_OBJECTS];
```

```
   logic           obj_active[MAX_OBJECTS];

  // tile
   localparam int TILE_COUNT = 8;
   logic [10:0] tile_x[0:7];
   logic [9:0]  tile_y[0:7];
   logic [5:0]  tile_index[0:7];


   // Sprite Drawing
   localparam int SPRITE_SIZE  = SPRITE_WIDTH * SPRITE_HEIGHT; //
16*16=256
   logic [13:0] sprite_address;
   logic [7:0] rom_data;
   logic [23:0] sprite_data;

   // star
   logic [6:0] frame_count_64;
   logic       star_bright_64;
   logic [6:0] frame_count_48;
   logic       star_bright_48;
   logic [6:0] frame_count_21;
   logic       star_bright_21;


   // ROM IP module
   //rom_sprites
   soc_system_rom_sprites sprite_images (
       .address      (sprite_address),
       .chipselect   (1'b1),
       .clk          (clk),
       .clken        (1'b1),
```

```
    .debugaccess  (1'b0),
    .freeze       (1'b0),
    .reset        (1'b0),
    .reset_req    (1'b0),
    .write        (1'b0),
    .writedata    (32'b0),
    .readdata     (rom_data)
);
soc_system_rom_s1 sprite_images1 (
    .address      (sprite_1_address),
    .chipselect   (1'b1),
    .clk          (clk),
    .clken        (1'b1),
    .debugaccess  (1'b0),
    .freeze       (1'b0),
    .reset        (1'b0),
    .reset_req    (1'b0),
    .write        (1'b0),
    .writedata    (32'b0),
    .readdata     (rom_1_data)
);

//color palette
logic [7:0] color_address_tile, color_address;
logic [23:0] color_data_tile, color_data;

assign color_address_tile = rom_1_data;
assign color_address  = rom_data;
color_palette palette_inst (
    .clk        (clk),
    .clken      (1'b1),
    .address    (color_address),
```

```
    .color_data (color_data)
);
color_palette palette_rom1 (
    .clk        (clk),
    .clken      (1'b1),
    .address    (color_address_tile),
    .color_data (color_data_1)
);
logic [13:0] sprite_1_address;
logic [7:0]  rom_1_data;
logic [7:0]  color_address_1;
logic [23:0] color_data_1;
assign color_data_tile = color_data_1;
assign sprite_data = color_data;

// Instantiate VGA counter module
vga_counters counters(.clk50(clk), .*);

// Register update logic
always_ff @(posedge clk) begin //initialize
    if (reset) begin

        background_r <= 8'h00;
        background_g <= 8'h80;
        background_b <= 8'h00;
        score <= 8'h00;

        for (int i = 0; i < MAX_OBJECTS; i++) begin
            obj_x[i] <= 12'd0;
            obj_y[i] <= 12'd0;
            obj_sprite[i] <= 6'd0;
            obj_active[i] <= 1'b0;
```

```
            end

        end

        else if (chipselect && write) begin
            case (address)
                5'd0: {background_r, background_g, background_b} <=
writedata[23:0];
                5'd1: score <= writedata[7:0];
                default: begin
                    if (address >= 7'd1 && address <= 7'd1 +
MAX_OBJECTS - 1) begin
                        int obj_idx;
                        obj_idx = address - 7'd1;
                        obj_x[obj_idx] <= writedata[31:20];
                        obj_y[obj_idx] <= writedata[19:8];
                        obj_sprite[obj_idx] <= writedata[7:2];
                        obj_active[obj_idx] <= writedata[1];
                    end
                end
            endcase
        end
    end

    // star
    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
        frame_count_64 <= 7'd0;
        end else if (hcount[10:1] == 11'd0 && vcount[9:0] == 10'd0)
begin
        frame_count_64 <= frame_count_64 + 1'b1;
        end
```

```
   end

   assign star_bright_64 = ~frame_count_64[6];


   always_ff @(posedge clk or posedge reset) begin
   if (reset)              frame_count_48 <= 7'd0;
   else if (hcount[10:1]==0 && vcount[9:0]==0) begin
       if (frame_count_48 == 7'd95) frame_count_48 <= 7'd0;
       else                         frame_count_48 <= frame_count_48
+ 1;
   end
   end

   assign star_bright_48 = (frame_count_48 < 7'd48);


   always_ff @(posedge clk or posedge reset) begin
   if (reset)              frame_count_21 <= 7'd0;
   else if (hcount[10:1]==0 && vcount[9:0]==0) begin
       if (frame_count_21 == 7'd41) frame_count_21 <= 7'd0;
       else                         frame_count_21 <= frame_count_21
+ 1;
   end
   end

   assign star_bright_21 = (frame_count_48 < 7'd21);

   // score
   logic [3:0] hundreds, tens, ones;
   logic [7:0] value;
   assign value = score;
```

```
   assign hundreds = value / 100;
   assign tens     = (value % 100) / 10;
   assign ones     = value % 10;

   // Stage00:
   always_comb begin
       tile_x[0] = 1280 - 9*SPRITE_WIDTH;  tile_y[0] = 0;
tile_index[0] = 10;
       tile_x[1] = 1280 - 8*SPRITE_WIDTH;  tile_y[1] = 0;
tile_index[1] = 11;
       tile_x[2] = 1280 - 7*SPRITE_WIDTH;  tile_y[2] = 0;
tile_index[2] = 12;
       tile_x[3] = 1280 - 6*SPRITE_WIDTH;  tile_y[3] = 0;
tile_index[3] = 13;
       tile_x[4] = 1280 - 5*SPRITE_WIDTH;  tile_y[4] = 0;
tile_index[4] = 14;
       tile_x[5] = 1280 - 3*SPRITE_WIDTH;  tile_y[5] = 0;
tile_index[5] = hundreds;
       tile_x[6] = 1280 - 2*SPRITE_WIDTH;  tile_y[6] = 0;
tile_index[6] = tens;
       tile_x[7] = 1280 - 1*SPRITE_WIDTH;  tile_y[7] = 0;
tile_index[7] = ones;
   end

   logic [6:0] active_obj_idx;
   logic obj_visible;
   logic found; //
   logic found_tile;
   logic tile_sprite;
   logic [3:0] rel_x, rel_y;
   logic [3:0] tile_rel_x, tile_rel_y;
```

```
logic [23:0] pix;
logic [23:0] pix_candidate; //
always_comb begin
    found = 1'b0;
    found_tile = 1'b0;
    obj_visible = 1'b0;
    active_obj_idx = 7'd0;
    rel_x = 4'd0;
    rel_y = 4'd0;
    tile_rel_y = 4'b0;
    tile_rel_x = 4'b0;
    pix             = {background_r,background_g,background_b};
    pix_candidate   = {background_r,background_g,background_b};
    sprite_address = 14'd0;
    sprite_1_address = 14'd0;
    tile_sprite =1'b1;
        // --- static star background ---
    if (star_bright_64 && (
        (hcount[10:1] == 654 && vcount[9:0] == 114) ||
        (hcount[10:1] == 25  && vcount[9:0] == 759) ||
        (hcount[10:1] == 281 && vcount[9:0] == 250) ||
        (hcount[10:1] == 228 && vcount[9:0] == 142) ||
        (hcount[10:1] == 754 && vcount[9:0] == 104) ||
        (hcount[10:1] == 692 && vcount[9:0] == 758) ||
        (hcount[10:1] == 558 && vcount[9:0] == 89)  ||
        (hcount[10:1] == 604 && vcount[9:0] == 432) ||
        (hcount[10:1] == 32  && vcount[9:0] == 30)  ||
        (hcount[10:1] == 95  && vcount[9:0] == 223) ||
        (hcount[10:1] == 238 && vcount[9:0] == 517) ||
        (hcount[10:1] == 616 && vcount[9:0] == 27)  ||
        (hcount[10:1] == 574 && vcount[9:0] == 203) ||
        (hcount[10:1] == 733 && vcount[9:0] == 665)
```

```
    )) begin
        pix = 24'hFFFFFF;  // white star when bright
    end

    if (star_bright_48 && (
        (hcount[10:1] == 718 && vcount[9:0] == 558) ||
        (hcount[10:1] ==  43 && vcount[9:0] == 517) ||
        (hcount[10:1] == 154 && vcount[9:0] ==  17) ||
        (hcount[10:1] == 320 && vcount[9:0] == 567) ||
        (hcount[10:1] == 602 && vcount[9:0] == 561) ||
        (hcount[10:1] == 369 && vcount[9:0] == 768) ||
        (hcount[10:1] == 707 && vcount[9:0] == 267) ||
        (hcount[10:1] ==  81 && vcount[9:0] == 326) ||
        (hcount[10:1] == 249 && vcount[9:0] == 618) ||
        (hcount[10:1] == 129 && vcount[9:0] == 608) ||
        (hcount[10:1] == 323 && vcount[9:0] == 142) ||
        (hcount[10:1] == 367 && vcount[9:0] == 164) ||
        (hcount[10:1] == 721 && vcount[9:0] == 440) ||
        (hcount[10:1] == 231 && vcount[9:0] == 322) ||
        (hcount[10:1] == 249 && vcount[9:0] == 598) ||
        (hcount[10:1] == 622 && vcount[9:0] == 599) ||
        (hcount[10:1] == 366 && vcount[9:0] == 282) ||
        (hcount[10:1] == 382 && vcount[9:0] == 646) ||
        (hcount[10:1] == 675 && vcount[9:0] == 472) ||
        (hcount[10:1] == 487 && vcount[9:0] == 307) ||
        (hcount[10:1] == 202 && vcount[9:0] == 596) ||
        (hcount[10:1] == 450 && vcount[9:0] == 770) ||
        (hcount[10:1] == 115 && vcount[9:0] == 152) ||
        (hcount[10:1] == 684 && vcount[9:0] ==  22)
    )) begin
        pix = 24'hFFFFFF;  // white star when bright
    end
```

```
if (star_bright_21 && (
    (hcount[10:1] == 684 && vcount[9:0] ==  22) ||
    (hcount[10:1] == 615 && vcount[9:0] == 512) ||
    (hcount[10:1] == 243 && vcount[9:0] == 159) ||
    (hcount[10:1] == 337 && vcount[9:0] == 527) ||
    (hcount[10:1] == 363 && vcount[9:0] == 216) ||
    (hcount[10:1] ==  60 && vcount[9:0] == 612) ||
    (hcount[10:1] == 354 && vcount[9:0] == 527) ||
    (hcount[10:1] ==  36 && vcount[9:0] == 488) ||
    (hcount[10:1] ==  13 && vcount[9:0] == 223) ||
    (hcount[10:1] == 491 && vcount[9:0] ==  18) ||
    (hcount[10:1] == 203 && vcount[9:0] == 171) ||
    (hcount[10:1] ==  28 && vcount[9:0] == 478) ||
    (hcount[10:1] == 585 && vcount[9:0] == 441) ||
    (hcount[10:1] == 438 && vcount[9:0] == 318) ||
    (hcount[10:1] == 214 && vcount[9:0] == 666) ||
    (hcount[10:1] == 300 && vcount[9:0] == 445) ||
    (hcount[10:1] == 161 && vcount[9:0] == 464) ||
    (hcount[10:1] ==   3 && vcount[9:0] == 739) ||
    (hcount[10:1] == 736 && vcount[9:0] == 269) ||
    (hcount[10:1] == 512 && vcount[9:0] == 780) ||
    (hcount[10:1] == 182 && vcount[9:0] == 519) ||
    (hcount[10:1] == 108 && vcount[9:0] == 640) ||
    (hcount[10:1] == 305 && vcount[9:0] == 654) ||
    (hcount[10:1] == 519 && vcount[9:0] == 623) ||
    (hcount[10:1] == 203 && vcount[9:0] == 156) ||
    (hcount[10:1] == 382 && vcount[9:0] == 780) ||
    (hcount[10:1] == 165 && vcount[9:0] == 552)
)) begin
    pix = 24'hFFFFFF;  // white star when bright
end
```

```
for (int i = TILE_COUNT - 1; i >= 0; i--) begin
    if (!found_tile &&
        hcount[10:1] >= tile_x[i][9:0] &&
        hcount[10:1] < tile_x[i][9:0] + TILE_WIDTH &&
        vcount[9:0] >= tile_y[i][9:0] &&
        vcount[9:0] < tile_y[i][9:0] + TILE_HEIGHT) begin
        tile_rel_x = hcount[10:1] - tile_x[i][9:0];
        tile_rel_y = vcount[9:0] - tile_y[i][9:0];
        sprite_1_address = tile_index[i] * SPRITE_SIZE
                         + tile_rel_y * SPRITE_WIDTH
                         + tile_rel_x;
        if (tile_rel_x == 0) begin
            pix_candidate = 24'h000000;
        end else begin
            pix_candidate = color_data_tile;
        end

        if (pix_candidate != 24'h000000) begin
            pix   = pix_candidate;
            found_tile = 1'b1;
        end
    end
end
tile_sprite = 1'b0;
if (!found_tile) begin
    for (int i = MAX_OBJECTS - 1; i >= 0; i--) begin
        if (!found &&
            obj_active[i] &&
            hcount[10:1] >= obj_x[i][9:0] &&
            hcount[10:1] < obj_x[i][9:0] + SPRITE_WIDTH &&
            vcount[9:0] >= obj_y[i][9:0] &&
```

```
                    vcount[9:0] < obj_y[i][9:0] + SPRITE_HEIGHT)
begin

                    active_obj_idx = i[6:0];
                    rel_x = hcount[10:1] - obj_x[i][9:0];
                    rel_y = vcount[9:0] - obj_y[i][9:0];
                    sprite_address = obj_sprite[active_obj_idx] *
SPRITE_SIZE
                                   + rel_y * SPRITE_WIDTH
                                   + rel_x;
                    // 原本 pix_candidate = sprite_data;
                    // 如果 rel_x==0，就把它当作透明色
                    if (rel_x == 0) begin
                        pix_candidate = 24'h000000;
                    end else begin
                        pix_candidate = sprite_data;
                    end
                    // 只用透明色判别
                    if (pix_candidate != 24'h000000) begin
                        pix   = pix_candidate;
                        found = 1'b1;
                    end
                end
            end
        end
        {VGA_R, VGA_G, VGA_B} = pix;
    end

endmodule

// VGA timing generator module
module vga_counters(
```

```
   input logic        clk50, reset,
   output logic [10:0] hcount,  // hcount是像素列，hcount[10:1]是实际显
示的像素位置
   output logic [9:0]  vcount,  // vcount是像素行
   output logic        VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n,
VGA_SYNC_n
);

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

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

   logic 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;
```

```
   logic endOfField;

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

   assign endOfField = vcount == VTOTAL - 1;

   // Horizontal sync: from 0x520 to 0x5DF (0x57F)
   assign VGA_HS = !( (hcount[10:8] == 3'b101) & !(hcount[7:5] ==
3'b111));
   assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

   assign VGA_SYNC_n = 1'b0; // For putting sync on the green
signal; unused

   // Horizontal active: 0 to 1279     Vertical active: 0 to 479
   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: rising edge
sensitive

endmodule
module color_palette(
   input  logic        clk,
   input  logic        clken,
   input  logic [7:0]  address,
   output logic [23:0] color_data
```

```
);
   always_ff @(posedge clk) begin
       if (clken) begin
           case (address)
               8'd0: color_data <= 24'h000000;
               8'd1: color_data <= 24'h000033;
               8'd2: color_data <= 24'h000066;
               8'd3: color_data <= 24'h000099;
               8'd4: color_data <= 24'h0000CC;
               8'd5: color_data <= 24'h0000FF;
               8'd6: color_data <= 24'h003300;
               8'd7: color_data <= 24'h003333;
               8'd8: color_data <= 24'h003366;
               8'd9: color_data <= 24'h003399;
               8'd10: color_data <= 24'h0033CC;
               8'd11: color_data <= 24'h0033FF;
               8'd12: color_data <= 24'h006600;
               8'd13: color_data <= 24'h006633;
               8'd14: color_data <= 24'h006666;
               8'd15: color_data <= 24'h006699;
               8'd16: color_data <= 24'h0066CC;
               8'd17: color_data <= 24'h0066FF;
               8'd18: color_data <= 24'h009900;
               8'd19: color_data <= 24'h009933;
               8'd20: color_data <= 24'h009966;
               8'd21: color_data <= 24'h009999;
               8'd22: color_data <= 24'h0099CC;
               8'd23: color_data <= 24'h0099FF;
               8'd24: color_data <= 24'h00CC00;
               8'd25: color_data <= 24'h00CC33;
               8'd26: color_data <= 24'h00CC66;
               8'd27: color_data <= 24'h00CC99;
```

```
                8'd28: color_data <= 24'h00CCCC;
                8'd29: color_data <= 24'h00CCFF;
                8'd30: color_data <= 24'h00FF00;
                8'd31: color_data <= 24'h00FF33;
                8'd32: color_data <= 24'h00FF66;
                8'd33: color_data <= 24'h00FF99;
                8'd34: color_data <= 24'h00FFCC;
                8'd35: color_data <= 24'h00FFFF;
                8'd36: color_data <= 24'h330000;
                8'd37: color_data <= 24'h330033;
                8'd38: color_data <= 24'h330066;
                8'd39: color_data <= 24'h330099;
                8'd40: color_data <= 24'h3300CC;
                8'd41: color_data <= 24'h3300FF;
                8'd42: color_data <= 24'h333300;
                8'd43: color_data <= 24'h333333;
                8'd44: color_data <= 24'h333366;
                8'd45: color_data <= 24'h333399;
                8'd46: color_data <= 24'h3333CC;
                8'd47: color_data <= 24'h3333FF;
                8'd48: color_data <= 24'h336600;
                8'd49: color_data <= 24'h336633;
                8'd50: color_data <= 24'h336666;
                8'd51: color_data <= 24'h336699;
                8'd52: color_data <= 24'h3366CC;
                8'd53: color_data <= 24'h3366FF;
                8'd54: color_data <= 24'h339900;
                8'd55: color_data <= 24'h339933;
                8'd56: color_data <= 24'h339966;
                8'd57: color_data <= 24'h339999;
                8'd58: color_data <= 24'h3399CC;
                8'd59: color_data <= 24'h3399FF;
```

```
                8'd60: color_data <= 24'h33CC00;
                8'd61: color_data <= 24'h33CC33;
                8'd62: color_data <= 24'h33CC66;
                8'd63: color_data <= 24'h33CC99;
                8'd64: color_data <= 24'h33CCCC;
                8'd65: color_data <= 24'h33CCFF;
                8'd66: color_data <= 24'h33FF00;
                8'd67: color_data <= 24'h33FF33;
                8'd68: color_data <= 24'h33FF66;
                8'd69: color_data <= 24'h33FF99;
                8'd70: color_data <= 24'h33FFCC;
                8'd71: color_data <= 24'h33FFFF;
                8'd72: color_data <= 24'h660000;
                8'd73: color_data <= 24'h660033;
                8'd74: color_data <= 24'h660066;
                8'd75: color_data <= 24'h660099;
                8'd76: color_data <= 24'h6600CC;
                8'd77: color_data <= 24'h6600FF;
                8'd78: color_data <= 24'h663300;
                8'd79: color_data <= 24'h663333;
                8'd80: color_data <= 24'h663366;
                8'd81: color_data <= 24'h663399;
                8'd82: color_data <= 24'h6633CC;
                8'd83: color_data <= 24'h6633FF;
                8'd84: color_data <= 24'h666600;
                8'd85: color_data <= 24'h666633;
                8'd86: color_data <= 24'h666666;
                8'd87: color_data <= 24'h666699;
                8'd88: color_data <= 24'h6666CC;
                8'd89: color_data <= 24'h6666FF;
                8'd90: color_data <= 24'h669900;
                8'd91: color_data <= 24'h669933;
```

```
            8'd92: color_data <= 24'h669966;
            8'd93: color_data <= 24'h669999;
            8'd94: color_data <= 24'h6699CC;
            8'd95: color_data <= 24'h6699FF;
            8'd96: color_data <= 24'h66CC00;
            8'd97: color_data <= 24'h66CC33;
            8'd98: color_data <= 24'h66CC66;
            8'd99: color_data <= 24'h66CC99;
            8'd100: color_data <= 24'h66CCCC;
            8'd101: color_data <= 24'h66CCFF;
            8'd102: color_data <= 24'h66FF00;
            8'd103: color_data <= 24'h66FF33;
            8'd104: color_data <= 24'h66FF66;
            8'd105: color_data <= 24'h66FF99;
            8'd106: color_data <= 24'h66FFCC;
            8'd107: color_data <= 24'h66FFFF;
            8'd108: color_data <= 24'h990000;
            8'd109: color_data <= 24'h990033;
            8'd110: color_data <= 24'h990066;
            8'd111: color_data <= 24'h990099;
            8'd112: color_data <= 24'h9900CC;
            8'd113: color_data <= 24'h9900FF;
            8'd114: color_data <= 24'h993300;
            8'd115: color_data <= 24'h993333;
            8'd116: color_data <= 24'h993366;
            8'd117: color_data <= 24'h993399;
            8'd118: color_data <= 24'h9933CC;
            8'd119: color_data <= 24'h9933FF;
            8'd120: color_data <= 24'h996600;
            8'd121: color_data <= 24'h996633;
            8'd122: color_data <= 24'h996666;
            8'd123: color_data <= 24'h996699;
```

```
                8'd124: color_data <= 24'h9966CC;
                8'd125: color_data <= 24'h9966FF;
                8'd126: color_data <= 24'h999900;
                8'd127: color_data <= 24'h999933;
                8'd128: color_data <= 24'h999966;
                8'd129: color_data <= 24'h999999;
                8'd130: color_data <= 24'h9999CC;
                8'd131: color_data <= 24'h9999FF;
                8'd132: color_data <= 24'h99CC00;
                8'd133: color_data <= 24'h99CC33;
                8'd134: color_data <= 24'h99CC66;
                8'd135: color_data <= 24'h99CC99;
                8'd136: color_data <= 24'h99CCCC;
                8'd137: color_data <= 24'h99CCFF;
                8'd138: color_data <= 24'h99FF00;
                8'd139: color_data <= 24'h99FF33;
                8'd140: color_data <= 24'h99FF66;
                8'd141: color_data <= 24'h99FF99;
                8'd142: color_data <= 24'h99FFCC;
                8'd143: color_data <= 24'h99FFFF;
                8'd144: color_data <= 24'hCC0000;
                8'd145: color_data <= 24'hCC0033;
                8'd146: color_data <= 24'hCC0066;
                8'd147: color_data <= 24'hCC0099;
                8'd148: color_data <= 24'hCC00CC;
                8'd149: color_data <= 24'hCC00FF;
                8'd150: color_data <= 24'hCC3300;
                8'd151: color_data <= 24'hCC3333;
                8'd152: color_data <= 24'hCC3366;
                8'd153: color_data <= 24'hCC3399;
                8'd154: color_data <= 24'hCC33CC;
                8'd155: color_data <= 24'hCC33FF;
```

```
            8'd156: color_data <= 24'hCC6600;
            8'd157: color_data <= 24'hCC6633;
            8'd158: color_data <= 24'hCC6666;
            8'd159: color_data <= 24'hCC6699;
            8'd160: color_data <= 24'hCC66CC;
            8'd161: color_data <= 24'hCC66FF;
            8'd162: color_data <= 24'hCC9900;
            8'd163: color_data <= 24'hCC9933;
            8'd164: color_data <= 24'hCC9966;
            8'd165: color_data <= 24'hCC9999;
            8'd166: color_data <= 24'hCC99CC;
            8'd167: color_data <= 24'hCC99FF;
            8'd168: color_data <= 24'hCCCC00;
            8'd169: color_data <= 24'hCCCC33;
            8'd170: color_data <= 24'hCCCC66;
            8'd171: color_data <= 24'hCCCC99;
            8'd172: color_data <= 24'hCCCCCC;
            8'd173: color_data <= 24'hCCCCFF;
            8'd174: color_data <= 24'hCCFF00;
            8'd175: color_data <= 24'hCCFF33;
            8'd176: color_data <= 24'hCCFF66;
            8'd177: color_data <= 24'hCCFF99;
            8'd178: color_data <= 24'hCCFFCC;
            8'd179: color_data <= 24'hCCFFFF;
            8'd180: color_data <= 24'hFF0000;
            8'd181: color_data <= 24'hFF0033;
            8'd182: color_data <= 24'hFF0066;
            8'd183: color_data <= 24'hFF0099;
            8'd184: color_data <= 24'hFF00CC;
            8'd185: color_data <= 24'hFF00FF;
            8'd186: color_data <= 24'hFF3300;
            8'd187: color_data <= 24'hFF3333;
```

```
            8'd188: color_data <= 24'hFF3366;
            8'd189: color_data <= 24'hFF3399;
            8'd190: color_data <= 24'hFF33CC;
            8'd191: color_data <= 24'hFF33FF;
            8'd192: color_data <= 24'hFF6600;
            8'd193: color_data <= 24'hFF6633;
            8'd194: color_data <= 24'hFF6666;
            8'd195: color_data <= 24'hFF6699;
            8'd196: color_data <= 24'hFF66CC;
            8'd197: color_data <= 24'hFF66FF;
            8'd198: color_data <= 24'hFF9900;
            8'd199: color_data <= 24'hFF9933;
            8'd200: color_data <= 24'hFF9966;
            8'd201: color_data <= 24'hFF9999;
            8'd202: color_data <= 24'hFF99CC;
            8'd203: color_data <= 24'hFF99FF;
            8'd204: color_data <= 24'hFFCC00;
            8'd205: color_data <= 24'hFFCC33;
            8'd206: color_data <= 24'hFFCC66;
            8'd207: color_data <= 24'hFFCC99;
            8'd208: color_data <= 24'hFFCCCC;
            8'd209: color_data <= 24'hFFCCFF;
            8'd210: color_data <= 24'hFFFF00;
            8'd211: color_data <= 24'hFFFF33;
            8'd212: color_data <= 24'hFFFF66;
            8'd213: color_data <= 24'hFFFF99;
            8'd214: color_data <= 24'hFFFFCC;
            8'd215: color_data <= 24'hFFFFFF;
            8'd216: color_data <= 24'h000000;
            8'd217: color_data <= 24'h2F5B89;
            8'd218: color_data <= 24'h5EB612;
            8'd219: color_data <= 24'h8D119B;
```

```
                8'd220: color_data <= 24'hBC6C24;
                8'd221: color_data <= 24'hEBC7AD;
                8'd222: color_data <= 24'h1A2236;
                8'd223: color_data <= 24'h497DBF;
                8'd224: color_data <= 24'h78D848;
                8'd225: color_data <= 24'hA733D1;
                8'd226: color_data <= 24'hD68E5A;
                8'd227: color_data <= 24'h05E9E3;
                8'd228: color_data <= 24'h34446C;
                8'd229: color_data <= 24'h639FF5;
                8'd230: color_data <= 24'h92FA7E;
                8'd231: color_data <= 24'hC15507;
                8'd232: color_data <= 24'hF0B090;
                8'd233: color_data <= 24'h1F0B19;
                8'd234: color_data <= 24'h4E66A2;
                8'd235: color_data <= 24'h7DC12B;
                8'd236: color_data <= 24'hAC1CB4;
                8'd237: color_data <= 24'hDB773D;
                8'd238: color_data <= 24'h0AD2C6;
                8'd239: color_data <= 24'h392D4F;
                8'd240: color_data <= 24'h6888D8;
                8'd241: color_data <= 24'h97E361;
                8'd242: color_data <= 24'hC63EEA;
                8'd243: color_data <= 24'hF59973;
                8'd244: color_data <= 24'h24F4FC;
                8'd245: color_data <= 24'h534F85;
                8'd246: color_data <= 24'h82AA0E;
                8'd247: color_data <= 24'hB10597;
                8'd248: color_data <= 24'hE06020;
                8'd249: color_data <= 24'h0FBBA9;
                8'd250: color_data <= 24'h3E1632;
                8'd251: color_data <= 24'h6D71BB;
```

```
                8'd252: color_data <= 24'h9CCC44;
                8'd253: color_data <= 24'hCB27CD;
                8'd254: color_data <= 24'hFA8256;
                8'd255: color_data <= 24'h29DDDF;
                default: color_data <= 24'h000000;
            endcase
        end
    end
endmodule
```