Flappy Bird on DE1-SoC Board

Giovanni Argueta

***Abstract*—The following report details the development of the reconstruction of the infamous Flappy Bird game on the DE1-SoC development board. Throughout our lab projects in Embedded Systems(ENEB 454), we have learned to use embedded c to configure memory mapping and driver modules to drive a various array of user-level programs. To complete these labs, understanding the system architecture and memory organization behind the DE1-SoC board was crucial, and changed accordingly to the drive module we read and wrote from. The final project prompt required the creation of a game with animation and the incorporation of multiple drivers. All reference codes and board information can be found in Intel's FPGA University Program DE1-SoC Computer manual [1].**

I. Introduction

This paper will follow the three phases that the project took to complete. Beforehand we must introduce the three drivers we implemented prior to the final project that was incorporated into our final user level c program. The first driver module configured prior worked with the HEX device driver. On the DE1 board, this HEX driver wrote digits to the seven-segment hex display. The second driver module, the KEY driver, drove the key push button's input available on the board. The last driver module that is the foundation of our program is the video driver. This video driver used the on-board VGA controller to display images through the video-out port, utilizing a pixel and character buffer. It can write specific colors for pixels at (x,y) coordinates, lines of pixels (using Bresenham's algorithm), boxes of pixels, write characters, and most importantly perform a pixel buffer swap by using the buffer controller to sync with the vertical sync cycle (1/60th of a second) of the VGA controller. This technique was very important in ensuring smooth animation.

The following figure displays the overview of the project based on the phase.

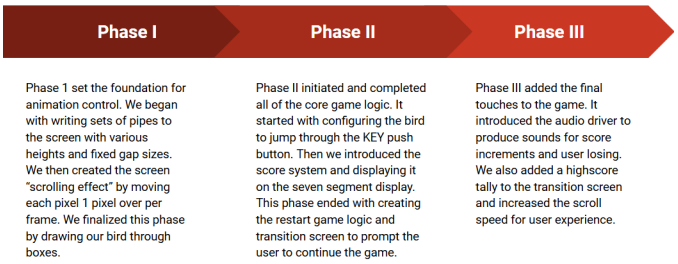

Fig. 1: Project overview

The following figure shows the hierarchy of our coding programs for this project. We will dive mostly in the content that the final.c file contains as it holds the game logic and communication with the video driver

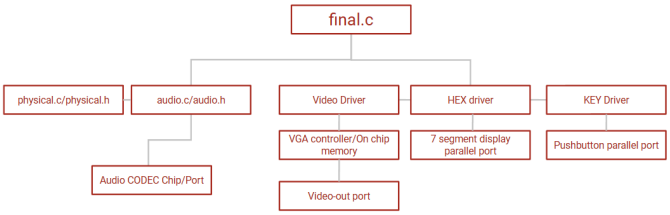

Fig. 2: Hierarchy Overview

II. Phase I

As mentioned Phase I involved setting the visual foundation for the program. We began with writing pipes to the screen. Initially, we randomized the height and gap between the pipes, but to ensure fairness for the user and replicate the original game, we fixed the gap size and limited the height size between every pair of pipes to around 10 pixels(the gap size vertically is also fixed). We also noticed the box would draw outside the boundaries so we used a utility function to ensure it fit within the constraints of the VGA screen. We use the pre-defined box command to draw the pipes. To introduce the scrolling mechanic, we introduced a new function that uses a scroll accumulator that adds fractional pixels for smooth scrolling. We decrement the pipes' x-coordinates by the accumulated amount (0.5 pixels per frame) and we ensure the pipes that move off-screen are reset to the right side with a new random height. After we iterate through our set of pipes (8), we call the draw_pipe to redraw each pipe. In the following figure, we highlight the scrolling pipe logic in the update_and_draw_pipes function.

```
// Move pipes left by SCROLL_SPEED pixels
scroll_accumulator += (float)SCROLL_SPEED / SCROLL_SPEED_MULTIPLIER;

// Only move pipes when we've accumulated at least 0.5 pixels of movement
if (scroll_accumulator >= 1.0f) {
    int pixels_to_move = (int)scroll_accumulator;
    scroll_accumulator -= pixels_to_move;

    for (int i = 0; i < MAX_PIPES; i++) {
        pipes[i].x -= pixels_to_move;
        if (pipes[i].x + PIPE_WIDTH < 0) {
            pipes[i].x = screen_x;
            pipes[i].top_height = MIN_PIPE_HEIGHT +
                rand() % (screen_y - GAP_SIZE - MIN_PIPE_HEIGHT);
        }
    }
}
```

Fig. 3: update_and_draw_pipes function

We then implemented the flappy bird visualization logic. We first started drawing the bird as three boxes, one as the body, one as the head, and one as the beak all as the color yellow. To imitate Flappy Bird's gravity effect, we use an accumulator

to move the bird down by accumulating a number of pixels (1) in every frame. We also make sure the bird does not fall out of the screen bounds. The following figures show the draw bird function and a portion of the update bird function.

```
void draw_bird(int fd) {
    // Draw body (rectangle)
    safe_draw_box(fd,
                  bird.x,
                  bird.y - BIRD_BODY_HEIGHT/2,
                  bird.x + BIRD_BODY_WIDTH,
                  bird.y + BIRD_BODY_HEIGHT/2,
                  BIRD_COLOR);

    // Draw head (square) - positioned at front of body
    safe_draw_box(fd,
                  bird.x + BIRD_BODY_WIDTH - BIRD_HEAD_SIZE/2,
                  bird.y - BIRD_BODY_HEIGHT/2 - BIRD_HEAD_SIZE/2,  // Position above body
                  bird.x + BIRD_BODY_WIDTH + BIRD_HEAD_SIZE/2,
                  bird.y - BIRD_BODY_HEIGHT/2 + BIRD_HEAD_SIZE/2,
                  BIRD_COLOR);

    // Draw beak (small square) - positioned at front of head
    safe_draw_box(fd,
                  bird.x + BIRD_BODY_WIDTH + BIRD_HEAD_SIZE/2,
                  bird.y - BIRD_BODY_HEIGHT/2 - BIRD_BEAK_SIZE/2,
                  bird.x + BIRD_BODY_WIDTH + BIRD_HEAD_SIZE/2 + BIRD_BEAK_SIZE,
                  bird.y - BIRD_BODY_HEIGHT/2 + BIRD_BEAK_SIZE/2,
                  BIRD_COLOR);
}
```

Fig. 4: draw_bird function

```
    // Update falling movement
    bird.fall_accumulator += (float)GRAVITY_SPEED / GRAVITY_MULTIPLIER;

    // Move bird down when we've accumulated enough for 1 pixel
    if (bird.fall_accumulator >= 1.0f) {
        int pixels_to_fall = (int)bird.fall_accumulator;
        bird.fall_accumulator -= pixels_to_fall;

        // Only update Y if not at bottom of screen
        if (bird.y + BIRD_BODY_HEIGHT/2 + pixels_to_fall < screen_y - BOTTOM_MARGIN) {
            bird.y += pixels_to_fall;
        }
    }
    // Keep bird within screen bounds
    if (bird.y - BIRD_BODY_HEIGHT/2 < 0) {
        bird.y = BIRD_BODY_HEIGHT/2;
    }
}
```

Fig. 5: update_bird function

To introduce jumping we use the input from the KEY driver(push button) so the bird moves a fixed amount of pixels up every time it was pressed.In this phase, we also opened and properly closed the video driver and KEY driver (upon exit) and implemented a signal handler for SIGINT so the user could exit the program using (Ctrl+C).

III. Phase II

As stated before Phase II involves all of the core logic behind the game mechanics. We first start with collision detection between the bird and the pipes. We treat the box as one whole rectangle, with four boundaries for left, right, top, and bottom (including the head and beak). Then the program loops through all of the pipes to see if there is an overlap with the pipe's horizontal range and if it is hitting the top or bottom pipe. The following figure shows the corresponding function.

```
// Check if bird has collided with any pipe
int check_collision() {
    // Get bird boundaries considering head and beak
    int bird_left = bird.x;
    int bird_right = bird.x + BIRD_BODY_WIDTH + BIRD_HEAD_SIZE/2 + BIRD_BEAK_SIZE;
    int bird_top = bird.y - BIRD_BODY_HEIGHT/2 - BIRD_HEAD_SIZE/2;  // Consider head position
    int bird_bottom = bird.y + BIRD_BODY_HEIGHT/2;

    // Check collision with each pipe
    for (int i = 0; i < MAX_PIPES; i++) {
        // If bird is within pipe's x-range
        if (!(bird_right < pipes[i].x || bird_left > pipes[i].x + PIPE_WIDTH)) {
            // Check if bird hits top pipe or bottom pipe
            if (bird_top < pipes[i].top_height ||  // Hit top pipe
                bird_bottom > pipes[i].top_height + GAP_SIZE) {  // Hit bottom pipe
                return 1;  // Collision detected
            }
        }
    }
    return 0;  // No collision
}
```

Fig. 6: check_collision function

Sequentially we wrote the logic to transition the user into a "Game over" screen where the user is prompted to restart. As mentioned before the video driver involves functionality to write to the char(character) buffer so the following function writes two lines to the middle of the screen, one of which prompts the user to press another push button (KEY1) to restart. The following figure shows such commands written to the video driver to drive the character writing.

```
// Function to display game over text
void display_game_over(int fd) {
    char command[64];


    snprintf(command, sizeof(command), "text %d,%d GAME OVER\n",
             GAME_OVER_X, GAME_OVER_Y);
    write(fd, command, strlen(command));

    snprintf(command, sizeof(command), "text %d,%d PRESS KEY1 to restart\n",
             RESTART_X, RESTART_Y);
    write(fd, command, strlen(command));
```

Fig. 7: display_game_over function

Afterward, we had to ensure the game properly restarted, so we introduced a new function that calls upon two new initialization functions for both pipes and the bird. It also resets all accumulators and scores (explained later) For the pipes, we initialize the pipes to the right edge of the screen(to give time for the user to move) and randomized the heights of the pipes within the maximum difference of 10 pixels. This function below is also called upon the start of the game (along with initializing the bird, which starts in the middle of the screen to the left)-

```c
// Function to initialize pipes with constrained height differences
void initialize_pipes() {
    srand(time(NULL));
    int start_x = screen_x;  // Start pipes off the right edge of the screen
    int previous_height = MIN_PIPE_HEIGHT + rand() % (screen_y - GAP_SIZE - MIN_PIPE_HEIGHT);

    for (int i = 0; i < MAX_PIPES; i++) {
        pipes[i].x = start_x + i * (PIPE_WIDTH + 60);  // Space pipes evenly to the right of the screen

        // Randomize height with a max difference constraint
        int min_height = previous_height - MAX_PIPE_HEIGHT_DIFF;
        int max_height = previous_height + MAX_PIPE_HEIGHT_DIFF;

        if (min_height < MIN_PIPE_HEIGHT) min_height = MIN_PIPE_HEIGHT;
        if (max_height > screen_y - GAP_SIZE) max_height = screen_y - GAP_SIZE;

        pipes[i].top_height = min_height + rand() % (max_height - min_height + 1);
        previous_height = pipes[i].top_height;  // Update previous height for the next pipe
    }
}
```

Fig. 8: initialize_pipes function

Next, we had to introduce our score system which by convention states the user scores 1 point for every pair of pipes fully passed by the bird. The new function checks if the bird's horizontal position has moved beyond the right edge of the pipe, and marks the pipe as "passed". The function mentioned can be seen here.

```c
void update_score() {
    for (int i = 0; i < MAX_PIPES; i++) {
        // Check if bird has fully passed this pipe and hasn't been counted yet
        if (!passed_pipes[i] &&
            bird.x > (pipes[i].x + PIPE_WIDTH)) {
            score++;
            passed_pipes[i] = 1;
            start_coin_sound(audio_virtual_base);
        }

        // Reset passed flag when pipe STARTS to wrap around, not after it's completely off screen
        if (pipes[i].x >= screen_x) {  // When pipe resets to right side
            passed_pipes[i] = 0;  // Reset flag so we can count it again
        }
    }
}
```

Fig. 9: update_score function

Instead of printing the score to the screen through characters, we decided to take advantage of the seven-segment display to update the score. We use the following function to use the hex driver's command to write its char buffer to then display the corresponding digit.

```c
void display_on_hex(int fd, int value) {
    char buffer[20];
    snprintf(buffer, sizeof(buffer), "%06d\n", value);
    write(fd, buffer, strlen(buffer));
}
```

Fig. 10: display_on_hex function

The last figure shows the hex display along with the game playing on the screen.

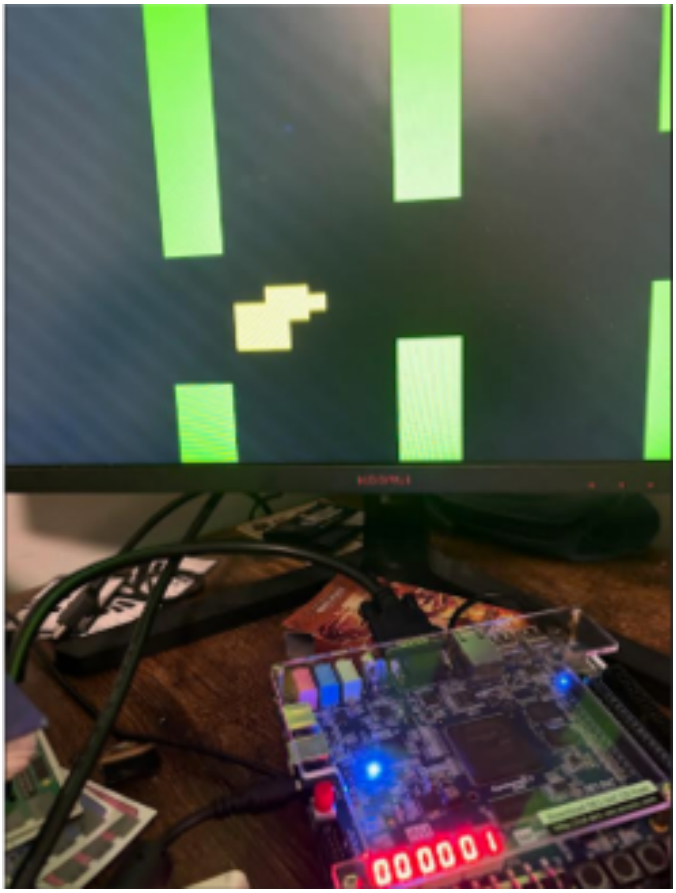

Fig. 11: Phase II results

IV. Phase III

As stated before, Phase III mainly consisted of configuring the audio driver. The audio port includes two write FIFOs (First In First Out) queues that are used to hold outgoing data and have a physical control register with the address 0xFF203040. We began with mapping the physical addresses by using Linux kernel functions such as sys/mman. Afterwards, to output audio we wrote sinusoidal waves to at certain hz tones to mimic the middle C chromatic scale and their frequencies. We needed to calculate the number of radians per sample for each tone and then use a loop to output a certain ms sample of each tone, given the sampling rate is 8,000 samples per sample. The figure shows three functions in audio.c, which allow the writing to the left and right channel registers, clearing of the audio FIFO buffers to reset any pending data, and ensuring synchronization by confirming the audio port has finished processing all data. Essentially they control the flow of audio flow.

Fig. 12: audio.c part 1

The following figure shows how we write our first sound (score sound that mimics a classic Super Mario coin effect). We first point to the base address of the memory-mapped audio registers, then we use the pre-defined frequencies of the two tones in the coin sound (B5 and E6) to calculate the angular frequencies for the tones by multiplying by 2pi and dividing by the sample rate. Then we loop over the samples for the first and second tones.

```
// Play the coin sound
int i;
double freq1_rad = COIN_FREQ1 * PI2 / SAMPLING_RATE;
double freq2_rad = COIN_FREQ2 * PI2 / SAMPLING_RATE;
int samples_per_tone = (SAMPLING_RATE * COIN_DURATION) / 1000;

// Play first tone (B5)
for (i = 0; i < samples_per_tone; i++) {
    int sample = (int)(MAX_VOLUME / 4 * sin(i * freq1_rad));
    write_audio_sample(audio_virtual_base, sample);
}

// Play second tone (E6)
for (i = 0; i < samples_per_tone; i++) {
    int sample = (int)(MAX_VOLUME / 4 * sin(i * freq2_rad));
    write_audio_sample(audio_virtual_base, sample);
}
```

Fig. 13: audio.c part 2

Then after initializing the audio in the final program along with the mapping and unmapping the physical registers in physical.c, we tested our audio implementation. Our first pressing issue occurred when the system claimed there was no such file or directory in regards to ”sys/mman.h”. The following figure shows how we fixed such errors. We began with retrieving the page size by using sysconf(_SC_PAGE_SIZE) and used PAGE_ALIGN to round down the base down to the nearest page boundary. We then calculated the offset by taking the base (0xFF203040) and page size (4096) and subtracted them to get 0x40. We also made sure the span (memory size) is adjusted to include the offset. We then used mmap to map the page-aligned base address for the corrected span and the returned pointer accounts for the offset which allows us to interact with the memory. In the unmap_physical function, we ensured the unmap operation also considered the offset. It is important to note in physical.c we also included the opening of the physical device ”/dev/mem” and functionally to close it.

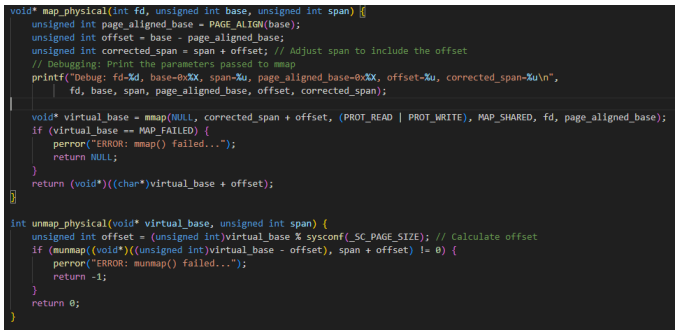

Fig. 14: physical.c part 1

This modification finally allowed us to successfully write audio to the port, and we used the same writing logic to add an ominous game over sound when the user enters the transition screen. We immediately noticed, however, that whenever these audios played, the program would freeze for about a frame or two. To combat this issue, we decided to introduce multi-threading. We begin with the use of pthread.h, a standardized API for thread creation and management. We first created the thread that contain both audios we introduced and used the audio virtual base pointer as an argument to the thread. We also make sure to detach the thread to clean up resources automatically when finishing without using pthread_join calls. Next, we initialized a mutex to ensure only one thread accesses the audio port at a time, so we lock this mutex before playing the audio and unlock it right after it is called upon. In summary, without threads, the main game loop would pause during audio playback, and by offloading audio processing to threads, the game can maintain its responsiveness. The following figure shows the initialization and locking/unlocking mechanism.

```
// Define the mutex
pthread_mutex_t audio_mutex = PTHREAD_MUTEX_INITIALIZER;

// Thread function for playing the coin sound
void* play_audio_thread(void* arg) {
    void* audio_virtual_base = arg;

    // Lock the mutex to ensure exclusive access to audio playback
    pthread_mutex_lock(&audio_mutex);

    // Unlock the mutex after playback
    pthread_mutex_unlock(&audio_mutex);
```

Fig. 15: thread/mutex creation

We finished this project by adding one final touch, the high score display on the game over screen. Similar to the real game, the real goal for the user is to beat their own high score so it proved useful to keep track of this score.

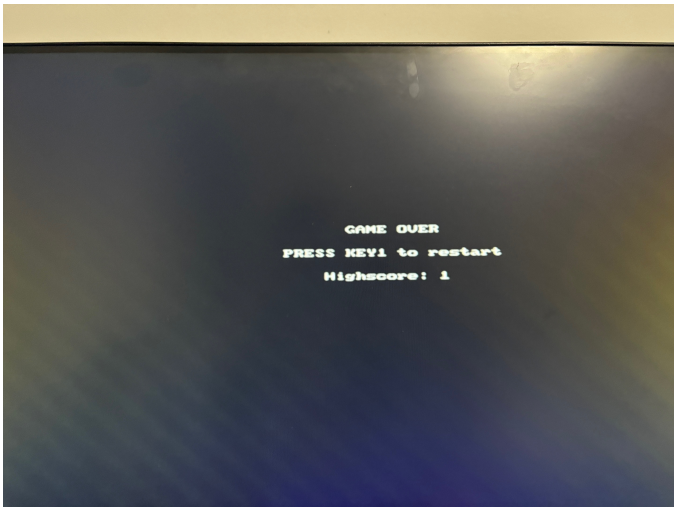

Fig. 16: game_over screen

V. Conclusion

Overall this project's progress went as expected, especially regarding incorporating the game logic. The lingering issue however is the limitation of the 0.5 pixel per frame scroll speed. When I tried to use a standard scroll speed such as 1 pixel per frame, a lot of jitteriness was introduced to the VGA screen. I tried multiple methods ranging from ensuring VGA buffer swap logic was sound, to implementing a command buffer so the user-level program would avoid writing too many commands per frame to draw pixels. This issue aside however I am glad I chose to incorporate the audio driver as it taught me how to account for page alignment for the physical base address of the device register for the audio port. This final project conveniently tied together many concepts we have learned over the course of the semester and it has made me interested in other embedded topics. I am particularly interested in machine learning with FPGAs as I want to learn how one can utilize hardware resource management capabilities to run algorithms on such devices.

VI. Appendix

Relevant Code:

```
// Initialize pipes with random positions and heights
initialize_pipes();
initialize_bird();

// Animation loop: update and redraw pipes until interrupted
printf("Starting main loop\n");
while (!stop) {
fflush(stdout);  // Ensure score is displayed immediately
update_and_draw_pipes(video_fd);  // Update positions and redraw pipes
display_on_hex(fd_hex, score);  // Update HEX display with current score
nanosleep(&frame_time, NULL);  // Maintain ~60 FPS
}

// Clear the screen before exiting
clear_text(video_fd);
write(video_fd, "clear_both\n", 10);
```

Fig. 17: Main loop of final.c

FPGA board inputs/outputs:

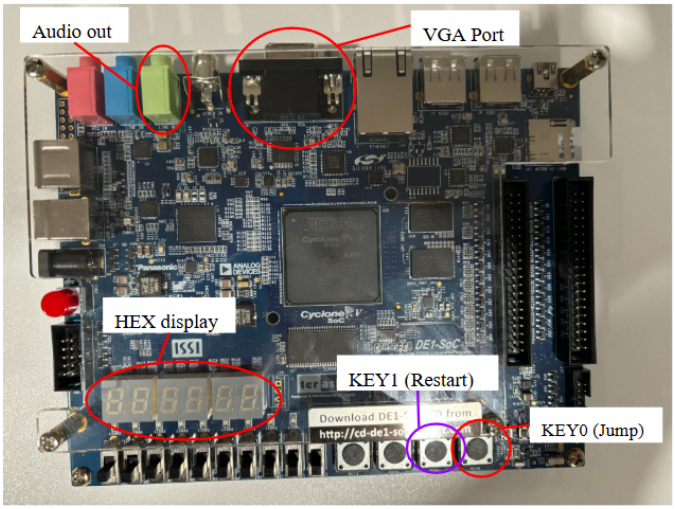

Fig. 18: DE1-SoC board inputs/outputs used

Please note the UART to USB port is not highlighted as it simply provided a serial connection to the device with no prior configuration. Additionally, the VGA port was connected to a VGA to HDMI converter as no monitor available had a VGFA port. For the audio, a speaker was connected directly to the audio out jack instead of through the monitor.

References

[1] Intel Corporation, DE1-SoC Computer System with Nios II User Manual, 2018. [Online]. Available: https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/Computer_Systems/DE1-SoC/DE1-SoC_Computer_NiosII.pdf.