The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

# ECE243 Final Group Project

A report describing the project
structure and how to run it on Altera's
DE1-SoC board

by

**Arnold Etta**[⋆] & **John David Genus**[†]

Submitted April 10, 2024

[⋆] Student ID: 1008745324, Email: arnold.etta@mail.utoronto.ca
[†] Student ID: 1008827101, Email: jd.genus@mail.utoronto.ca

**Abstract**

For our project, we built a game called FRUIT SMASH (see Figure 1). The idea behind this game originates from the "Candy Crush" game. In Candy Crush, players try to attain target scores or missions within a limited time frame or swap amount. Candies could only be swapped between candies immediately around themselves. A match is attained when 3 or more candies of the same type align with each other in either the vertical and/or horizontal alignment; whereby, matched candies break into points and the gaps are filled with more candies thereafter.

In our version, we will be making use of fruits instead of candies. There will be 6 fruits and infinite levels. Fruits will not have to be swapped between neighbouring fruits; so, players can select any two fruits on the screen to swap them. We have made this change to reduce the complexity of the game implementation and increase player freedom. To mitigate this change, we have made 2 adjustments deviating from the original implementation in Candy Crush:

- A match can only be made **horizontally** between 3 or more fruits
- Once a match of 3 or more fruits is attained, the fruits are destroyed (they disappear) and the locations they were previously occupying cannot be used for swapping

This will require players to employ more intelligent matching strategies to attain target scores. For the first level, players will be required to reach a target score of 15000 within 90 seconds. Next levels will have the target scores increased and time decreased.

Players make use of the PS2 keyboard to move a cursor to select fruits for swapping. Scores and time remaining are displayed on the VGA screen through the character buffer. The game time is measured using the Timer Device of the DE1-SoC board and background music plays through the audio out port of the board.


Figure 1: FRUIT SMASH start screen with game instructions

# Table of Contents

# List of Figures



# List of Tables

# Main Content

## 1. Introduction - Program Overview

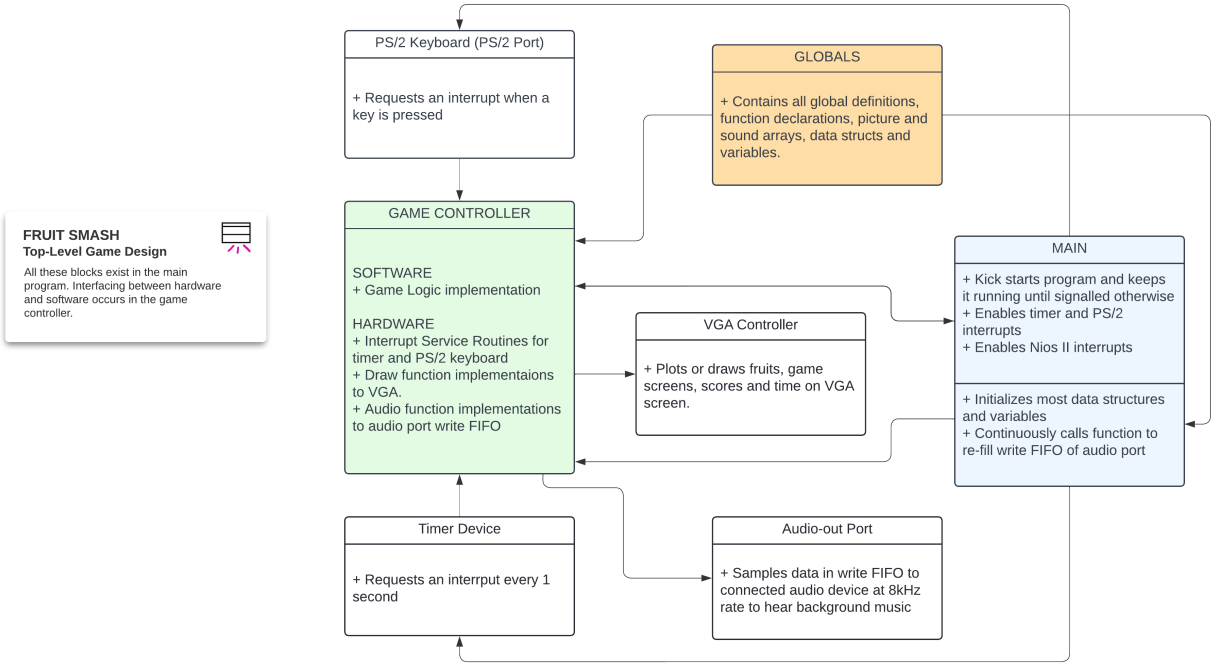


Figure 2: Top-level Overview of Fruit Smash game implementation

### 1.1. Data Structures

Almost all functions and supplementary variables were built around 2 structs of type *fruit* (Figure 3a) and *box_selector* (Figure 3b). Both structs have an x-y coordinate corresponding to their location on the game board. The *color_id* variable is used to locate a horizontal matching of 3 or more fruits (we call this a mesh). The boolean variables *isSmashed* and *isLatched* are used by the game controller to determine what game logic to implement; these include: swapping fruits, activating second box selector and erasing fruit from game board.

```
struct fruit{
    int x_pos, y_pos;
    int color_id;
    short int fruit_data[FRUIT_SIZE];

    bool isSmashed; // Initialize to 0. Push to 1 when fruit is part of a match mesh.
};
```

(a) Fruit struct with data variables

```
struct box_selector{
    int x_loc, y_loc;

    bool isLatched; // Initialize to 0. Push to 1 when user presses enter to select a fruit
};
```

(b) Box selector struct with data variables

Figure 3: Struct declarations with which game was built around

48 instances of the *fruit* struct was declared with every 8 instance corresponding to a particular fruit (out of the 6 possible fruits). 2 instances of the *box_selector* struct was declared. The 48 fruits were arranged into a 2D array using a sorting algorithm (6 rows, 8 cols) called *GMBrd*. [Figure 4].

```
struct fruit* GMBrd[BOARD_LENGTH][BOARD_WIDTH]; //Game Board (hopefully)

// 6 different types of Fruits. A maximum 8 of each fruit can appear on the Game Board.
struct fruit apple[8];
struct fruit orange[8];
struct fruit lemon[8];
struct fruit watermelon[8];
struct fruit blueberry[8];
struct fruit grape[8];

// 2 instantiated box selectors
struct box_selector sel_1;
struct box_selector sel_2;
```

Figure 4: Struct instantiations and *GMBrd* declaration

## 1.2. Enabling Nios II Interrupts

We made use of the macro functions provided by Intel's Altera to write into and read from the status, ienable and ipending control registers of the Nios II processor.

```
//2024.03.30.AE -> Original Version

//Author: Intel FPGA (DE1-SoC Computer System with Nios® II)


#ifndef __NIOS2_CTRL_REG_MACROS__
#define __NIOS2_CTRL_REG_MACROS__

/*****************************************************************************/
/* Macros for accessing the control registers.                               */
/*****************************************************************************/

    #define NIOS2_READ_STATUS(dest) \
        do { dest = __builtin_rdctl(0); } while (0)
    #define NIOS2_WRITE_STATUS(src) \
        do { __builtin_wrctl(0, src); } while (0)
    #define NIOS2_READ_ESTATUS(dest) \
        do { dest = __builtin_rdctl(1); } while (0)
    #define NIOS2_READ_BSTATUS(dest) \
        do { dest = __builtin_rdctl(2); } while (0)
    #define NIOS2_READ_IENABLE(dest) \
        do { dest = __builtin_rdctl(3); } while (0)
    #define NIOS2_WRITE_IENABLE(src) \
        do { __builtin_wrctl(3, src); } while (0)
    #define NIOS2_READ_IPENDING(dest) \
        do { dest = __builtin_rdctl(4); } while (0)
    #define NIOS2_READ_CPUID(dest) \
        do { dest = __builtin_rdctl(5); } while (0)
#endif
```

Figure 5: Nios II control register macro functions

# 2. Game Controller

This part of the game structure is responsible for taking in hardware input, executing game logic based on that input, updating the game board before sending to hardware output to display updated game. Thus, this section can be boiled down to two subsections being Game Logic and Hardware Handling.

## 2.1. Game Logic

We followed a basic game flow present in many infinite level games to implement the game logic (Figure 6). The crucial functions in this logic are the *sort gameboard* and *check match* algorithms. These functions effectively play the move and consequently update the game state after the move is played. Their implementation will be discussed in this subsection.

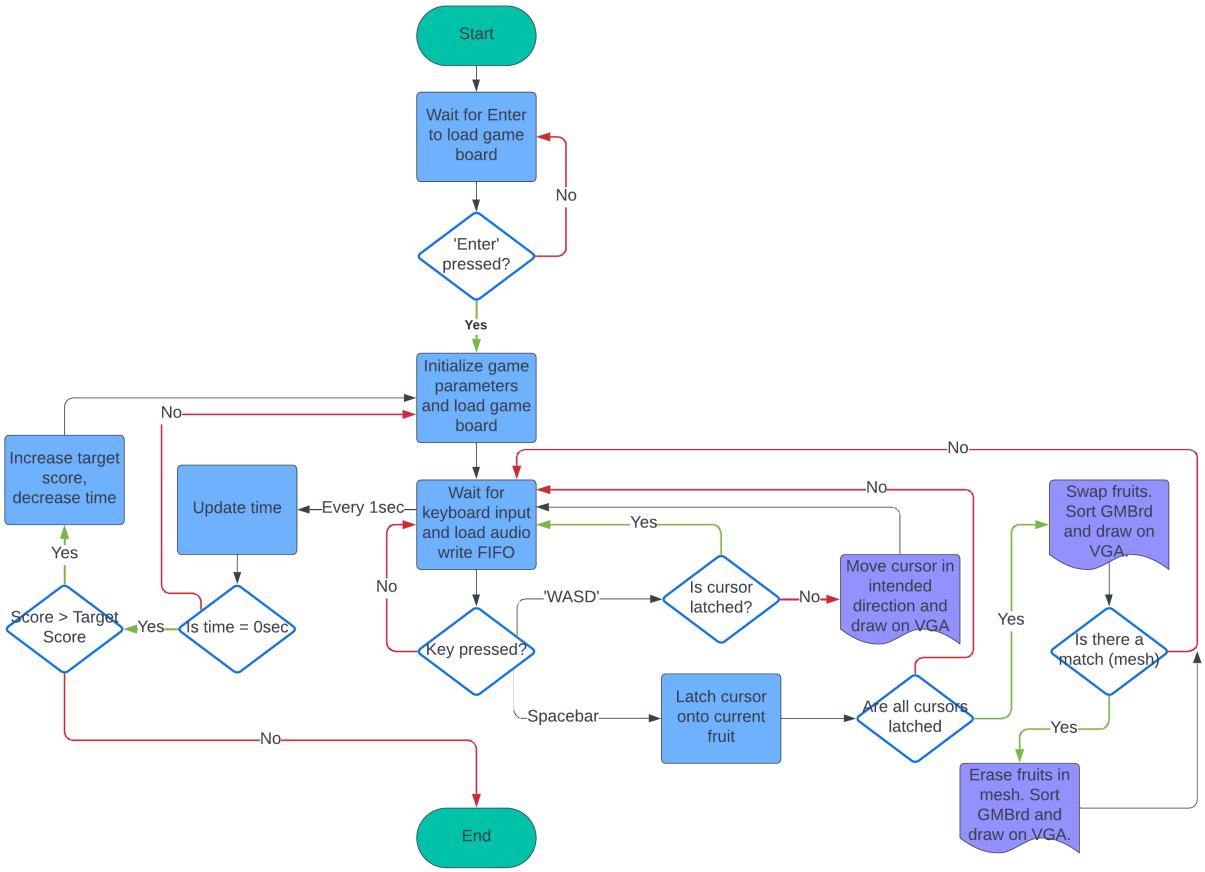


Figure 6: FRUIT SMASH game logic illustrated as a flow chart

### 2.1.1. Sort Game Board Algorithm

This algorithm was implemented to allow for an optimized method of searching for fruits in the 2D *GMBrd* array in O(1) complexity. By arranging fruits in the 2D array based on their x-y coordinate

(increasing y down the array and increasing x to the right), one would only need a simple relationship to between the x-y coordinate to know the location of the fruit in the array.

```
void sort_GMBrd(struct fruit* GMBrd[][BOARD_WIDTH]){
    // Flatten the 2D array into a 1D array for sorting
    struct fruit* flattened_grid[48];
    int k = 0;
    for (int j = 0; j < 6; j++) {
        for (int i = 0; i < 8; i++) {
            flattened_grid[k++] = GMBrd[j][i];
        }
    }

    sort_coordinates(flattened_grid);
    rearrange_grid(GMBrd, flattened_grid);
    sort_grid_by_x(GMBrd);
}
```

Figure 7: Game board sorting algorithm

As depicted in figure 7 above, we first flatten the 2D array into a 1D array of 48 elements. Then a bubble sort algorithm is used to sort the flattened array based on y-coordinates. Then the flattened array is transformed back to a 2D array. At this point we will see that from the top row to the bottom row, the y-coordinate value will be decreasing. Then each row is individually sorted using bubble sort based on their x-coordinates. This is the fully realized game board sorting algorithm.

With this function in place, all it would take to search for a fruit in the array based on their x-y coordinate will be to create a look-up table shown in figure 8.

```
struct fruit* search_xyCoord(struct fruit* GMBrd[][BOARD_WIDTH], int x, int y){

    switch (x*y)
    {
        case 2720:
            return GMBrd[0][0];
        case 3616:
            return GMBrd[0][1];
        case 4512:
            return GMBrd[0][2];
        case 5408:
            return GMBrd[0][3];
        case 6304:
            return GMBrd[0][4];
        case 7200:
            return GMBrd[0][5];
        case 8096:
            return GMBrd[0][6];
```

Figure 8: Part of O(1) look-up table used to search for a fruit based on x-y coordinate

### 2.1.2. Check Match Algorithm

Implemented a function that checks for any consecutive matches between 3 or more fruits and returns the number of fruits in the match mesh. This number is then used to calculate and update the player's score. [Figure 9]

```
int check_match(struct fruit* GMBrd[][BOARD_WIDTH]){
    int fruit_smashed = 0;
    //Check horizontal matching
    for(int row = 0; row < 6; row++){
        for(int col = 0; col < 8; col++){
            int count = 0;
            for(int step = 0; col + step < 8; step++){
                if((GMBrd[row][col+step]->color_id != GMBrd[row][col]->color_id)){
                    step = 8;
                }else if((GMBrd[row][col+step]->isSmashed == 0)){
                    count++;
                }
            }
            if(count >= 3){
                for(int i = 0; i < count; i++){
                    if(GMBrd[row][col+i]->isSmashed != 1){
                        GMBrd[row][col+i]->isSmashed = 1;
                        GMBrd[row][col+i]->offGMBrd = 1;
                    }else{
                        count;
                    }

                }
                fruit_smashed += count;
            }
        }
    }

    return fruit_smashed;
}
```

Figure 9: Check match function to determine fruits to erase from game board

Each fruit has a *color_id* integer and *isSmashed* boolean variable that is used to identify consecutive fruits and fruits that were already erased from the game board. This is important because erasing a fruit does not actually remove it from the *GMBrd* array but simply tells the draw functions to draw an empty space in it's supposed location. With these two variables in place, we start indexing from an element and step in to the next two elements to determine a matching *color_id*. If the next element's *color_id* does not match, we start indexing again from the next element. If the next element's *color_id* matches, we continue indexing until we the next element fails to match or until the end of the row. This ensures we are only checking for matches horizontally.

## 2.2. Hardware Handling

FRUIT SMASH makes use of a total of four hardware devices with two being inputs and two being outputs:

1. PS/2 Keyboard (Input)
2. Timer Device (Input)
3. VGA (Output)
4. Audio out Port (Output)

All input devices were handled through interrupts and outputs implemented wherever necessary throughout the program.

### 2.2.1. PS/2 Keyboard

To enable keyboard interrupts the following bit fields were written into:

- RE bit field of PS/2 control register
- Bit 7 of ienable control register

The interrupt service routine (ISR) for the keyboard starts off with extracting the 3 key codes at the head of the PS/2 FIFO. The RAVAIL field in the data register and a limiter variable to ensure only 3 keycodes are read [Figure 10a]. We then look at the last keycode read and use it to implement logic.

Before leaving the PS/2 ISR, the PS/2 FIFO is cleared so that we do not immediately re-enter the interrupt handler after leaving it [Figure 10b]. The RI bit field of the PS/2 control register goes to 0 if the FIFO is empty. So, the bit continuously polled whilst the data register is read from until it reaches 0.

```
while(RAVAIL != 0 && limiter < 3){
    PS2_Data = *ps2_ptr & 0xFFFF80FF;
    RAVAIL = PS2_Data & 0xFFFF0000;

    key_code_1 = key_code_2;
    key_code_2 = key_code_3;
    key_code_3 = PS2_Data & 0x000000FF;

    limiter++;
    printf("Keycode sent \n");
}
```

(a) Section of the PS/2 ISR that extracts keycodes

```
void clear_ps2_fifo(int PS2_data_reg){
    PS2_data_reg = *ps2_ptr & 0xFFFF80FF;
    int RI_field = *(ps2_ptr+1) & 0x00000100;
    while(RI_field != 0){
        PS2_data_reg = *ps2_ptr & 0xFFFF80FF;
        RI_field = *(ps2_ptr+1) & 0x00000100;
    }
}
```

(b) Clear FIFO function called before returning from ISR

Figure 10: Main components of PS/2 Keyboard Interrupt Handler

### 2.2.2. Timer Device

The DE1-SoC board has 2 interval timers out of which one timer device was made use of. To enable timer interrupts the following bit fields were written into:

- ITO bit field of Interval Timer control register
- Bit 0 of ienable control register

The timer device was setup to continuously interrupt the program every 1 second by loading 100,000,000 (100MHz clock) into the counter start registers and turning on the START and CONT bits of the control register [Figure 11a].

Once the program is interrupted by the timer device, we first write a 0 into the TO bit field of the status register to clear the interrupt [Figure 11b]. Then game logic is implemented thereafter before leaving the interrupt.

```
int counter = 100000000; // 1/(100 MHz) x (100_000_000) = 1000 msec or 1sec
*(interval_timer_ptr + 0x2) = (counter & 0xFFFF);
*(interval_timer_ptr + 0x3) = (counter >> 16) & 0xFFFF;
*(interval_timer_ptr + 1) = 0x7; // STOP = 0, START = 1, CONT = 1, ITO = 1
```

(a) Enable timer count down from 100,000,000

```
void interval_timer_ISR(void){
    *(interval_timer_ptr) = 0;

    timeRemaining -= 1;
```

(b) Clear timer interrupt before implementing game logic

Figure 11: Main components of Timer Device Interrupt Handler

### 2.2.3. VGA

There are two parts of the VGA which we utilized being the pixel buffer and the character buffer. The character buffer was used to draw scores, time and texts on the VGA screen while the pixel buffer was used to display the rest of the game [Figure ]. For a seamless fruit swapping and cursor movement animation, FRUIT SMASH was implemented using double buffering and optimized drawing functions.


Figure 12: Typical game screen with fruits on game board and scores displayed

We followed the standard setup for double buffering taught in Lab 7 but in addition to that, we created vga support functions [Figure 13a] and relentlessly optimized these functions [Figure 13b]. These functions now served as the building blocks used to create other draw functions such as *draw_fruit*, *draw_player_score* and so on.

```
/*VGA Support Function Declarations*/
void plot_pixel(int x, int y, short int line_color); // To draw a pixel to the screen
void write_char(int x, int y, char c); // To draw a single (8px by 8px) character to VGA screen
void draw_box(int x, int y, int x_size, int y_size, short int box_colour, bool fill);
void draw_screen(const short int* screen_array); // To a draw a 320x240 pic on the VGA screen
void clear_screen(void); // To paint entire screen black (may not use it at all)
void wait_for_vsync(void);
```

(a) VGA support functions

```
void draw_screen(const short int* screen_array){
    for(int y = 0; y < 240; y++){
        for(int x = 0; x < 320; x+=320){
            plot_pixel(x, y, screen_array[320*y + x]);
            plot_pixel(1, y, screen_array[320*y + 1]);
            plot_pixel(2, y, screen_array[320*y + 2]);
            plot_pixel(3, y, screen_array[320*y + 3]);
```

(b) Optimized function to draw 320x240 images faster

Figure 13: Enhancing VGA animation performance outside of double buffering

### 2.2.4. Audio out Port

The game was intended to have background music playing all through. We setup the audio port registers as done in Prof. Moshovos' notes and used a polling process provided to us by a TA to continuously fill the write FIFO [Figure 14].

```
struct audio_t {
      volatile unsigned int control;
      volatile unsigned char rarc;
      volatile unsigned char ralc;
      volatile unsigned char wsrc;
      volatile unsigned char wslc;
      volatile unsigned int ldata;
      volatile unsigned int rdata;
};

struct audio_t *const audiop = ((struct audio_t *)0xff203040);

struct audio_clip {
    // Info about song trying to play
    int* samples;
    int num_samples;
    // Where am I currently in the song
    int pos;
    // Flags
    bool valid; // is this currently a valid song
    bool restart; // should I restart when done
};
```

Figure 14: Audio support declarations and setup

In the main function, a while loop is run infinitely and in the loop, the *fillSoundQueue* function is continuously polled to load the audio FIFO [Figure 15a]. Loading only occurs when the number of samples in the write FIFO is below a certain threshold [Figure 15b].

```
play_sound(background_soundtrack, sizeof(background_soundtrack), true);
while(1){
    fillSoundQueue();
}
```

(a) While loop continuously polling fill function

```
void fillSoundQueue() {
    // While there is space in the audio queue
    int num_played_samples = 0;
    while (audiop->wsrc != 0 && num_played_samples < 10000) {
        // Build up the sample we want to play
        int sample = 0;
        bool foundSample = false;
        // Go over all currently playing audio clips and superimpose their samples together into
        sample
        for (int i =0 ; i < MAX_SOUNDS; i++) {
            if (sounds[i].valid) {
                sample += sounds[i].samples[sounds[i].pos];
                sounds[i].pos++;
```

(b) Section of function that fills write FIFO

Figure 15: Playing background music through audio out port

## 3. How to run FRUIT SMASH on Monitor Program

Turn on the DE1-SoC board and open quartus monitor program. Make a new project and go through the basic setting configuration described in the Monitor Program Tutorial up until the specify program details section. Include two files here:

1. address_map_nios2.h

2. fruitsmash_CPUlator.c

Then when you get to the specify memory settings section, choose change from .basic to .exceptions. Download the .sof file to the board, connect the keyboard to the DE1-SoC board, compile and load project, then run the project. Enjoy!

# 4. Attribution Table

| Section | Arnold | John David | Altera / ECE243 Staff |
|---|---|---|---|
| Introduction | ✓ | ✓ | - |
| Data Structures | ✓ | ✓ | - |
| Enabling Nios II Interrupts | - | - | ✓ |
| Sort Game Board Algorithm | ✓ | - | - |
| Check Match Algorithm | - | ✓ | - |
| PS/2 Keyboard | ✓ | - | - |
| Timer Device | - | ✓ | - |
| VGA | ✓ | - | ✓ |
| Audio out Port | - | ✓ | ✓ |

Table 1: Attribution Table.