7583842 Henry Chang
8405037 Ci-Chian Lu

ECE253 Final Project – NEXYSnake

I. Purpose

This game is the final project for ECE253 of Fall 2021. Our goal is to create a snake game powered by the FPGA board, Digilent NEXYS A7-100T (which is where the eponymous game name NEXYSnake comes from), with the game screen displayed on a HiLetgo 2.8" TFT LCD instead of an external monitor via VGA cable to reduce the uncertainty of timing issues. We combined lab assignments experiences and the concepts of finite state machines and scheduling from lectures to implement a smooth-running, delicate, yet easy-to-operate game.

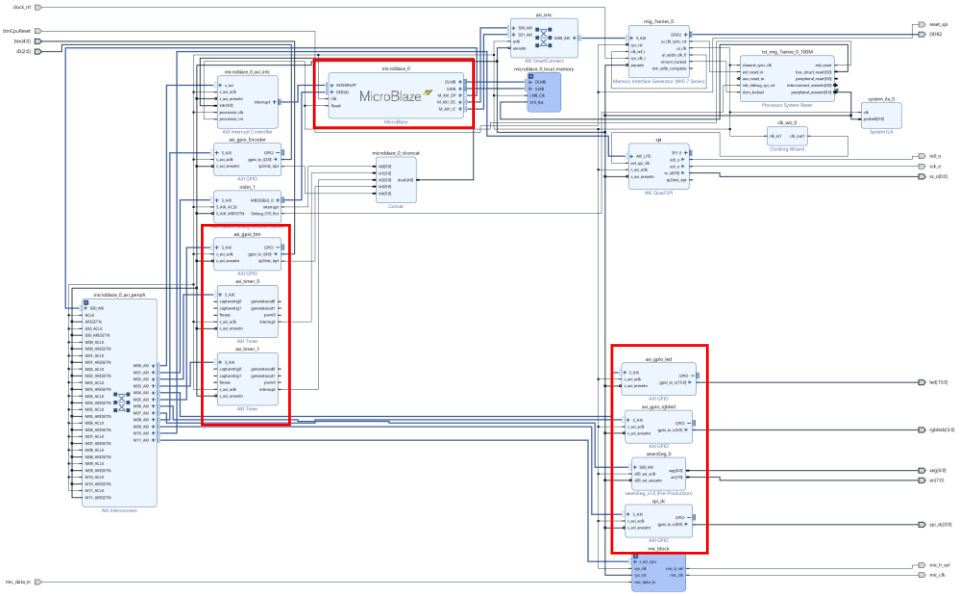
II. Methodology

The idea of the snake game is to maneuver the snake operable in four directions: upwards, downwards, leftwards, and rightwards within a bounded region. The snake grows in length and accelerates whenever it captures a randomly spawned apple in the area. The goal is to avoid collision with the four boundaries (north/south/east/west) and the snake itself.

The game provides feedback to alert the user of the situation or the state occurring to create a smooth gaming experience. The seven-segment display integrated on the FPGA displays texts such as "paused" and "you lose" while the game is paused and over respectively. The LCD pops out hints to inform users what operations should be performed to move on to the next step.

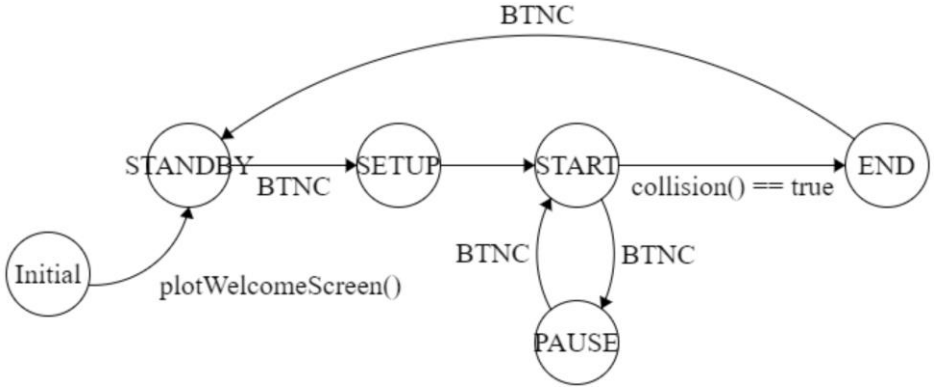
III. Results

a. Xilinx Vivado Block Diagram



Framed functional blocks are essential for this project. For the sake of convenience, we used the block diagram design of Lab 3A without alternation.

b. State Machine of NEXYSnake and Button Interrupt



Before entering the grand loop, the welcome screen is plotted on LCD (*plotWelcomeScreen()*). After entering the grand loop, the game will be in *STANDBY*, which is a pseudo-state that doesn't perform anything, but instead awaits the user to press *BTNC* on the FPGA to initiate the game. As the *BTNC* interrupt fires, the game enters *SETUP* to generate the initial settings for gameplay. The transition to *START* is automatic after game setup. The state machine will alternate between *START* and *PAUSE* depending on whether *BTNC* interrupts fire or not. If the user loses the game, i.e., *collision() == true*, the game arrives in *END*. It awaits the user to press *BTNC* again to return to *SETUP* and restart a new round.

```
init_method(); // initialize board and peripherals
plotWelcomeScreen();

/* Grand Loop */
while (true) {
    switch (gameState) {
        case STANDBY: clearSevenSeg(); break;
        case SETUP:   init_game();     break;
        case START:   startGame();     break;
        case PAUSE:   pauseGame();     break;
        case END:     gameOver();      break;
    }
}
```

While *BTNC* is used for state transition, the rest of the four buttons (*BTNU/BTNL/BTNR/BTND*) are only operable in *START*, which are used to control the orientation of snake in gameplay. The button functions are defined in the button interrupt handler.

```
switch (btn_value) {
    case BTNU:
        if (snakeDir != UP && snakeDir != DOWN && gameState == START) snakeDir = UP;
        break;
    case BTNL:
        if (snakeDir != LEFT && snakeDir != RIGHT && gameState == START) snakeDir = LEFT;
        break;
    case BTNR:
        if (snakeDir != RIGHT && snakeDir != LEFT && gameState == START) snakeDir = RIGHT;
        break;
    case BTND:
        if (snakeDir != DOWN && snakeDir != UP && gameState == START) snakeDir = DOWN;
        break;
    case BTNC:
        // set state
        switch (gameState) {
            case STANDBY:
                gameState = SETUP;
                xil_printf("State: SETUP\n\r");
                break;
            case START:
                printPlayPauseStop(gameState = PAUSE);
                xil_printf("State: PAUSE\n\r");
                break;
            case PAUSE:
                printPlayPauseStop(gameState = START);
                xil_printf("State: START\n\r");
                break;
            case END:
                gameState = STANDBY;
                xil_printf("State: STANDBY\n\r");
                plotWelcomeScreen();
                break;
        }
        break;
}
```

c. Game Algorithms

i. Snake position: The snake is declared by an array of structure *Snake*, storing the x and y coordinates of each body sections of the snakes. At the start of game, the default snake length is set to 5. The location of the snake head and the orientation of snake body is determined with a random function seeded by the value of timer register.

```
/*** generate initial snake position ***/
srand(XTmrCtr_GetTimerCounterReg(XPAR_TMRCTR_0_BASEADDR, 1));
snake[0].x = (rand() % (FRAMEWIDTH - snakeSize * 2) + snakeSize) * GRIDSIZE;
snake[0].y = (rand() % (FRAMEHEIGHT - snakeSize * 2) + snakeSize) * GRIDSIZE;

switch (rand() % 4) {
    case UP: // up
        for (int i = 1; i < snakeSize; ++i) {
            snake[i].x = snake[i - 1].x;
            snake[i].y = snake[i - 1].y - GRIDSIZE;
        }
        snakeDir = DOWN;
        break;
    case LEFT: // left
        for (int i = 1; i < snakeSize; ++i) {
            snake[i].x = snake[i - 1].x - GRIDSIZE;
            snake[i].y = snake[i - 1].y;
        }
        snakeDir = RIGHT;
        break;
    case RIGHT: // right
        for (int i = 1; i < snakeSize; ++i) {
            snake[i].x = snake[i - 1].x + GRIDSIZE;
            snake[i].y = snake[i - 1].y;
        }
        snakeDir = LEFT;
        break;
    case DOWN: // down
        for (int i = 1; i < snakeSize; ++i) {
            snake[i].x = snake[i - 1].x;
            snake[i].y = snake[i - 1].y + GRIDSIZE;
        }
        snakeDir = UP;
        break;
}
```

ii. Snake motion: The motion of snake can be represented as a series of array elements shifting excluding the first one (snake head). The new location of snake head is updated by adding or subtracting the x and y coordinates depending on snake's orientation.

```
void snakeMotion(void) {
    // shift snake body coordinates
    for (int i = snakeSize; i > 0; --i)
        snake[i] = snake[i - 1];

    // update snake head coordinate
    switch (snakeDir) {
        case UP:    snake[0].y -= GRIDSIZE; break;
        case LEFT:  snake[0].x -= GRIDSIZE; break;
        case RIGHT: snake[0].x += GRIDSIZE; break;
        case DOWN:  snake[0].y += GRIDSIZE; break;
    }

    return;
}
```

iii. Snake speed: The speed of snake is highly relevant with the update frequency of body coordinates. To prevent an abrupt snake sprint on LCD, *snakeSpeed()* is designed to restrict the time interval between the movement of snake from current location to the next location. The parameter *s* ranges from 0 to 20, which means the stopping time interval of Microblaze ranges from 200 to 400 milliseconds.

```
void snakeSpeed(u16 s) {
    MB_Sleep(400 - s * 10); // millisecond

    return;
}
```

iv. Food position: Food is spawned at the start of the game and each time after the food is eaten by snake. The new coordinate of food is generated with a random function seeded by the value of timer register. To avoid overlapping with snake, the generated coordinate needs comparison with the snake array. Food location will be regenerated if overlap occurs.

```
void spawnFood(void) {
    food.x = (rand() % FRAMEWIDTH) * GRIDSIZE;
    food.y = GRIDSIZE + (rand() % (FRAMEHEIGHT - 1)) * GRIDSIZE;

    // avoid overlapping with snake
    for (int i = 0; i < snakeSize; ++i) {
        if (food.x != snake[i].x || food.y != snake[i].y)
            continue;

        spawnFood();
        break;
    }

    return;
}
```

v. Food eaten: If the coordinate of the snake head is identical with that of the food, the food is determined as eaten.

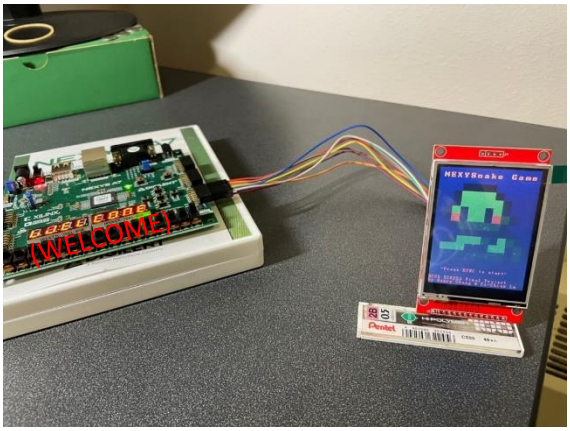
```
bool foodEaten(void) {
    if ((snake[0].x == food.x) && (snake[0].y == food.y))
        return true;

    return false;
}
```

vi. Collision: If the coordination of snake head exceeds the boundary, i.e., $x < 0 \,||\, x > 240 \,||\, y < 20 \,||\, y > 320$, it counts as border collision. If the snake head overlaps any part of the snake body, it counts as body collision.

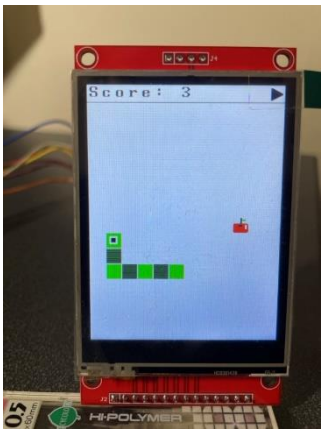
```
bool collision(void) {
    // border collision
    switch (snakeDir) {
        case UP:
            if (snake[1].y == GRIDSIZE) return true;
            break;
        case LEFT:
            if (snake[1].x == 0) return true;
            break;
        case RIGHT:
            if (snake[0].x == 240) return true;
            break;
        case DOWN:
            if (snake[0].y == 320) return true;
            break;
    }

    // body collision
    for (int i = 1; i < snakeSize; ++i) {
        if ((snake[i].x == snake[0].x) && (snake[i].y == snake[0].y))
            return true;
    }

    return false;
}
```

d. User Interface and Designs

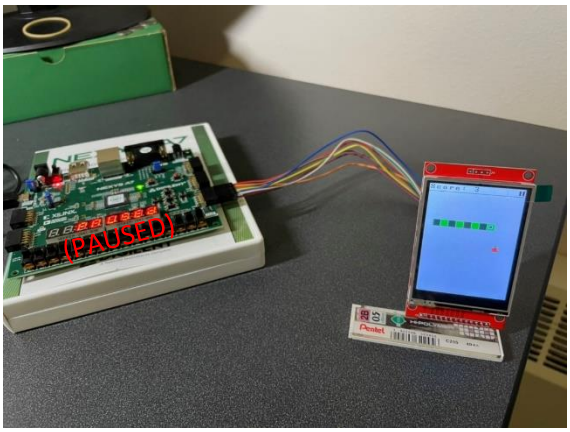
i. Welcome screen



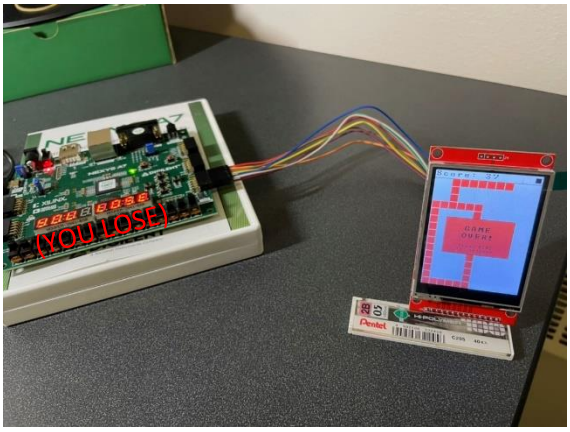
ii. Gameplay start



iii. Gameplay pause



iv. Game over



e. Issues: The update frequencies of LCD and seven-segment displays are different. A seven-segment display needs refreshing continuously since only one digit is displayable at once, while LCD needs only setting up once to draw on the canvas. To make both peripherals operate simultaneously, seven-segment display functions are placed within the grand loop, while LCD functions are instead mainly handled by button interrupts since LCD does not require refreshing periodically. However, snake and food plot functions are placed within the grand loop since the two objects require updating every cycle.

In our original proposal, we planned to build an UART interface between two FPGAs for communication. However, due to the conflict of register addresses, even the UART example on Xilinx's official GitHub cannot function. We assumed the two possible reasons might be: first, Digilent's peripheral hardware is noncompatible with Xilinx's library. Second, since there are three UART ports on the FPGA (MicroUSB port/UART by Xilinx/ UART-Lite by Xilinx), we cannot assure which will be activated when the board is programmed.