ELC 5396 System on a Chip: Final Project

Ryan Morrison

December 10, 2023

The Idea

Ear training is an essential tool for any musician trying to improve their craft. However, this process takes many years and quickly becomes monotonus. The idea behind this project is to develop an ear training system using the Nexys4 board that is challenging and fun.

The Game

The hardware for this project is quite simple: the Nexys4 board, a portable speaker that connects to the board via aux cable, and a keyboard that connects to the board via USB.

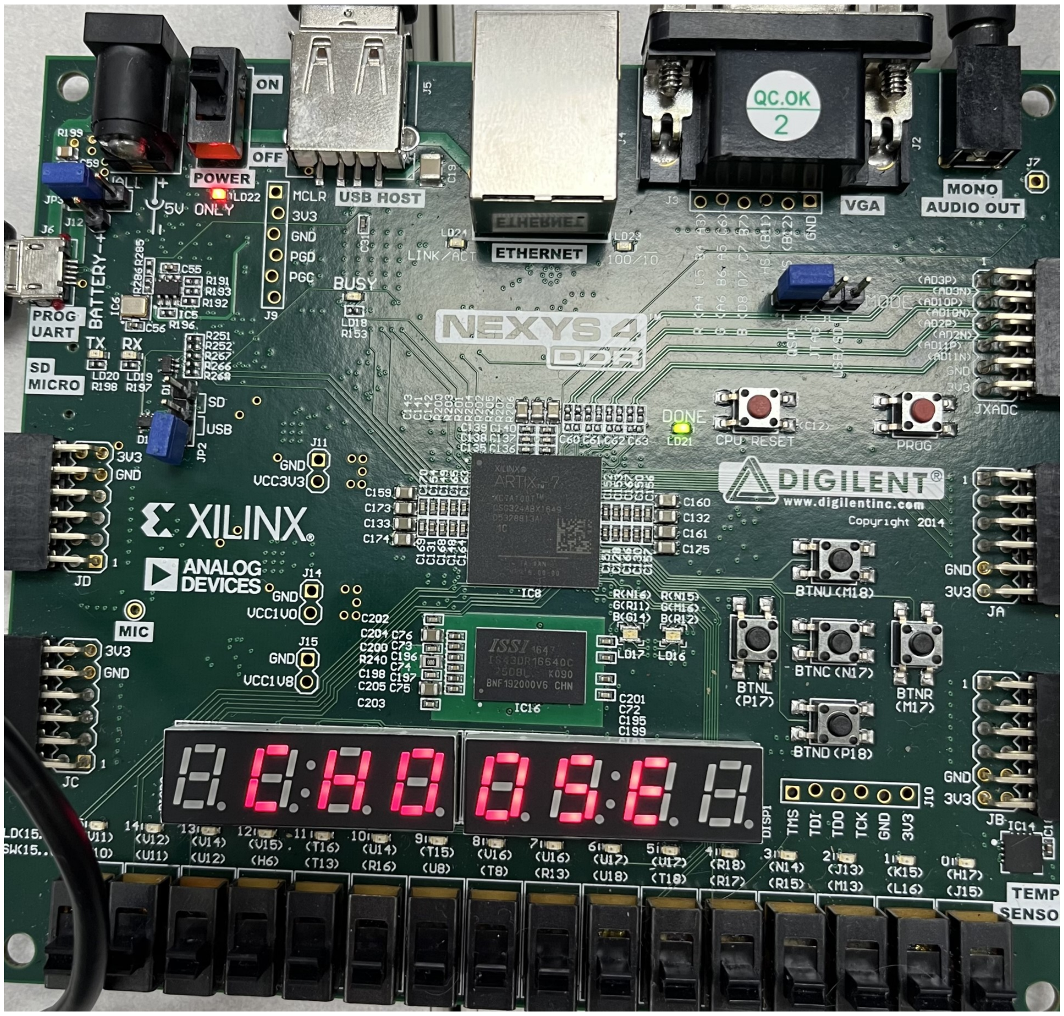


Figure 1: The Nexys4 Board

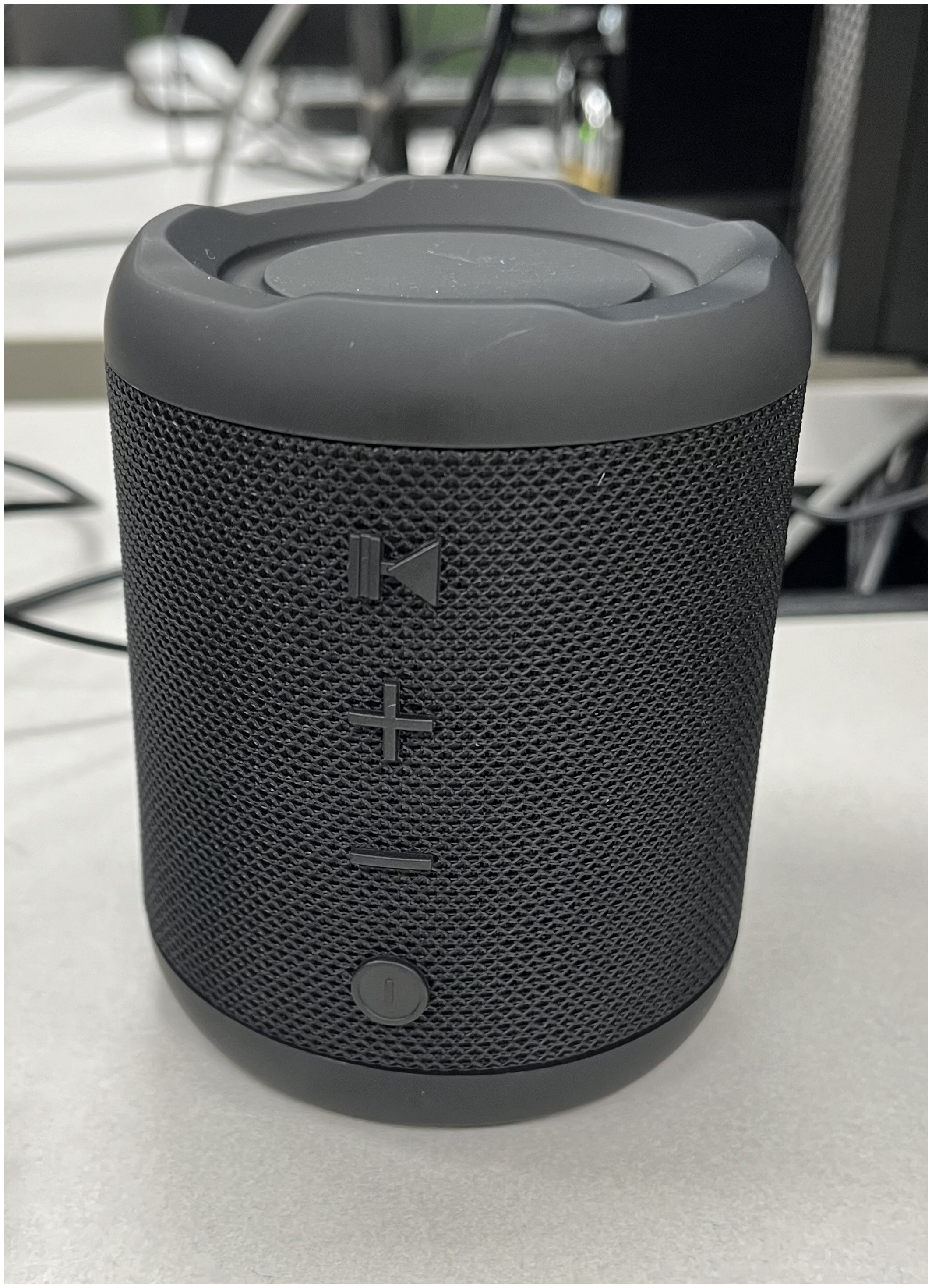


Figure 2: The Portable Speaker

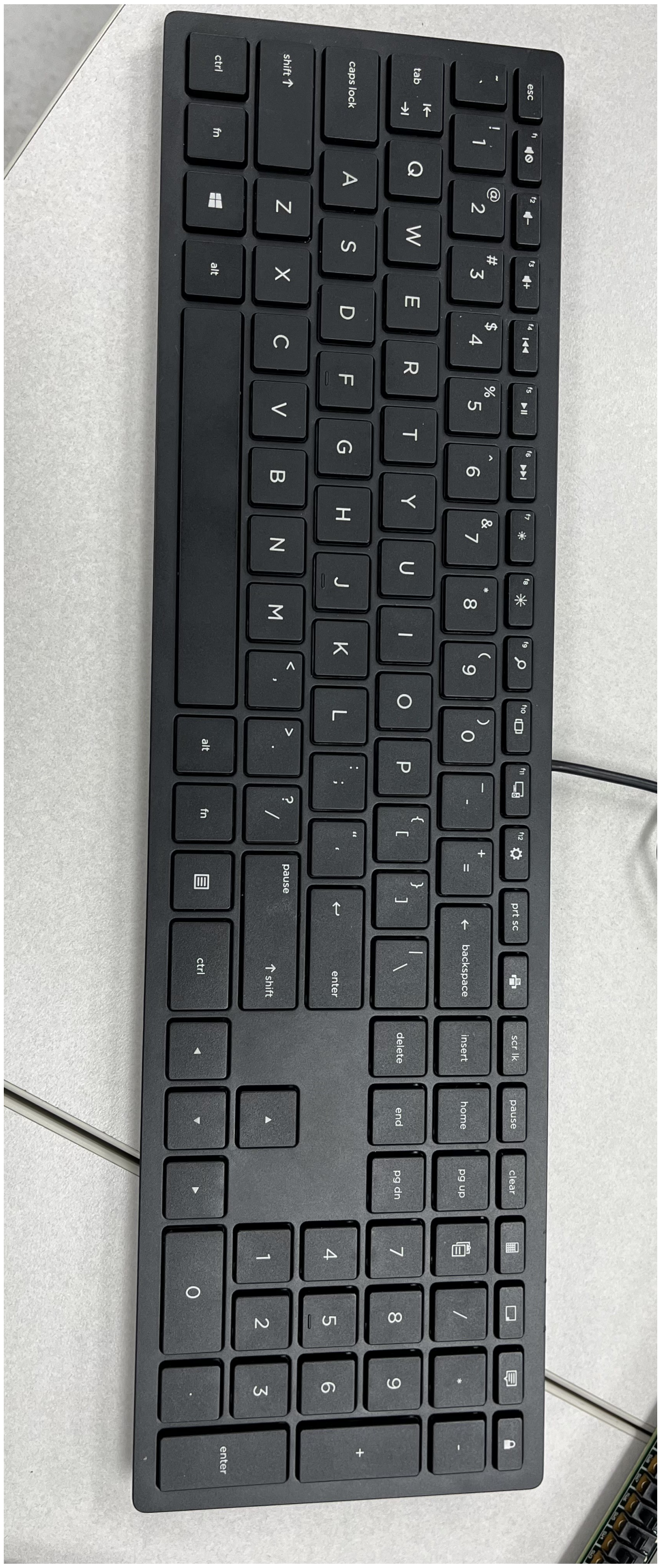


Figure 3: The Keyboard

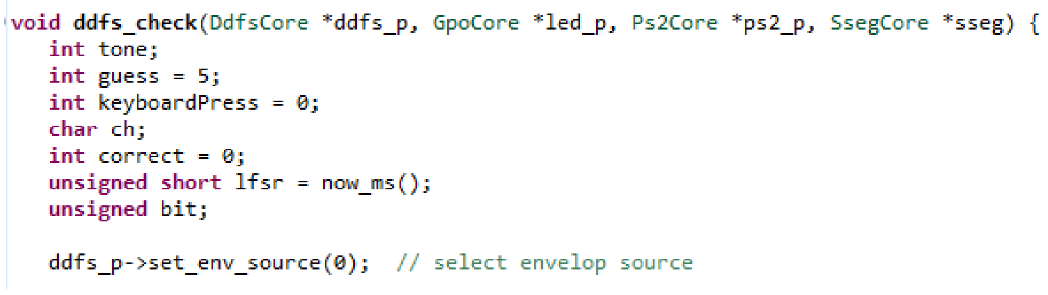
The Nexys4 board will randomly select one of three notes to play (C4, F4, or B4), and the speaker will play that note accordingly. Then, the player will press the corresponding key on the keyboard to state what they think the note is. For example, if they think the speaker played a C4, they would press the "c" key on the keyboard. If the player is correct, the Nexys4 board will let them know via the seven-segment display on the board. The board will also let the player know if they are wrong through the same process. The board will also keep track of how many correct notes the player gets; if they get five notes in a row correct, they've won the game. If they get a note wrong at any point though, the counter will reset.

The Process and Code

All of the software was written in Vitis, based off the Verilog bitstream that was generated earlier in the semester. This bitstream file is included in the GitHub repository linked here along with all other relevant files.

https://github.com/ryan2298/CR1/tree/main/FinalProject

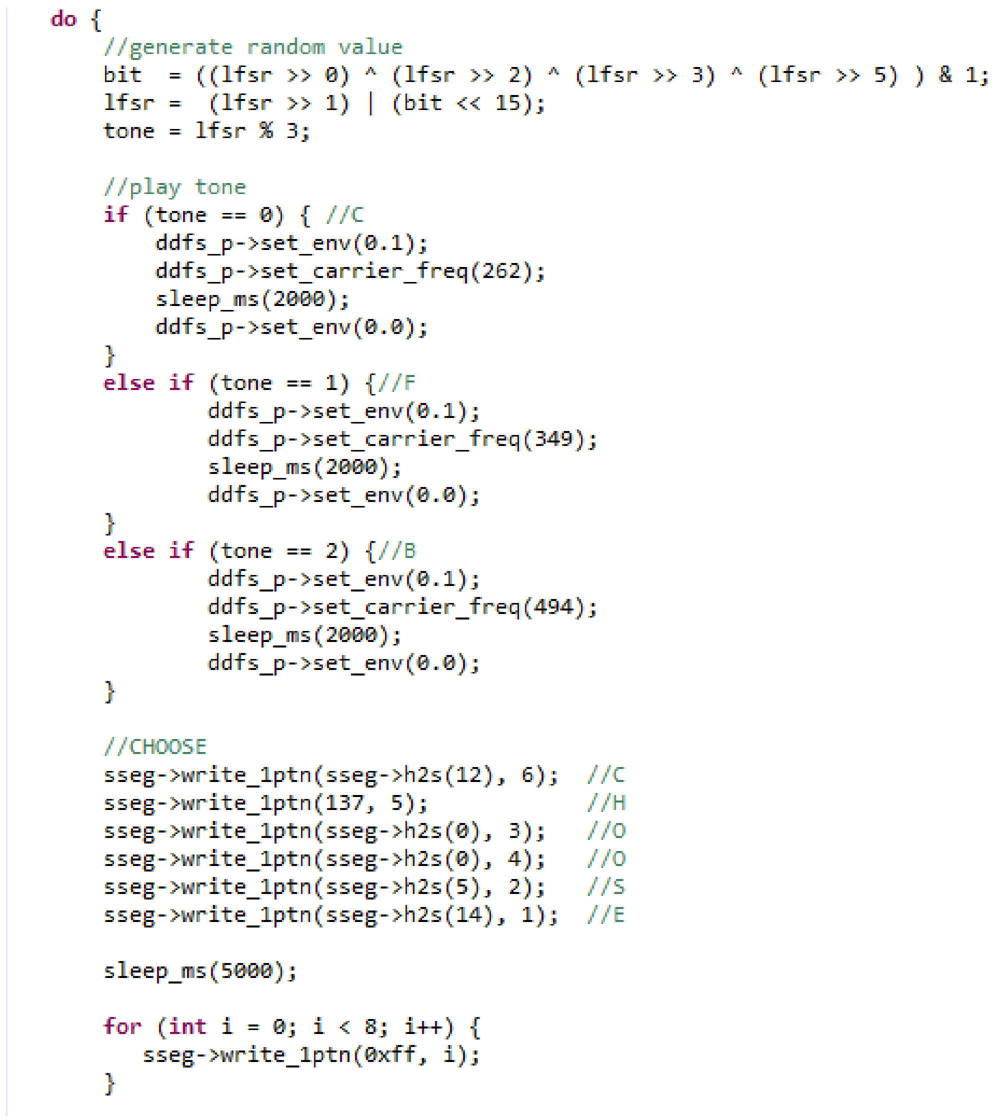
The Vitis code is based off of Chu's main-vanilla-test.cpp file and the following cores: SsegCore, Ps2Core, and DdfsCore. Chu's ddfs-check function is the backbone of the project, and it was edited to meet the specific needs of the project.



```
void ddfs_check(DdfsCore *ddfs_p, GpoCore *led_p, Ps2Core *ps2_p, SsegCore *sseg) {
   int tone;
   int guess = 5;
   int keyboardPress = 0;
   char ch;
   int correct = 0;
   unsigned short lfsr = now_ms();
   unsigned bit;

   ddfs_p->set_env_source(0);  // select envelop source
```

Figure 4: First Part of the Code

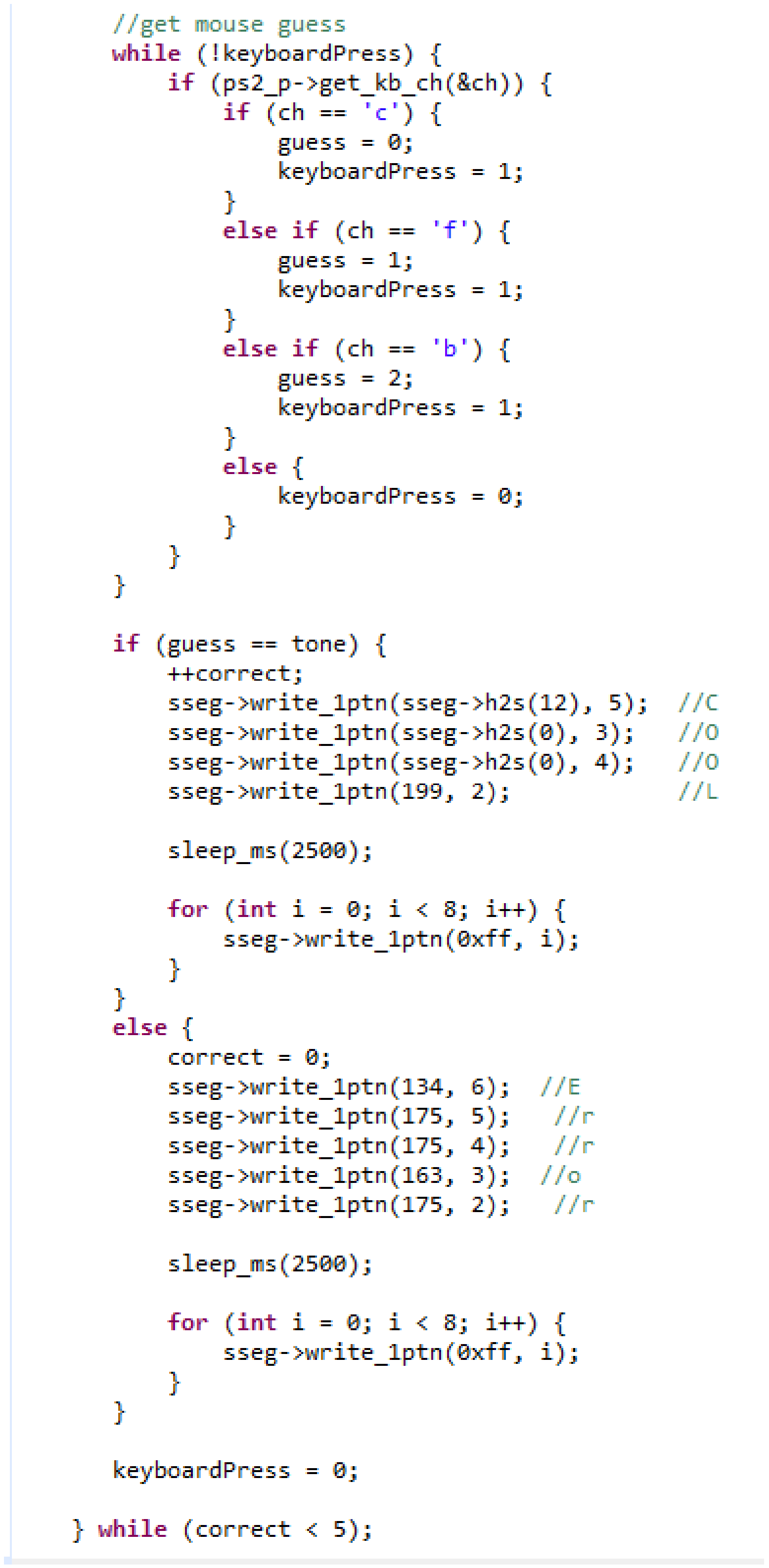
This first section is where the necessary parameters get passed in, and all the relevant variables are declared. The passed parameters are the necessary cores written by Chu. The tone integer will hold the randomly generated note to be played. The guess integer will hold the note selected by the player. The keyboardPress integer allows the program to know when a keyboard button has been pressed and therefore does not have to wait on the player input any longer. The ch character will hold the keyboard button pressed by the player. The correct integer will keep track of how many correct notes the player has guessed. The lfsr and bit variables are used for random note generation. The lfsr variable is based off of now-ms(), meaning that it will be a different value everytime the code is run. This ensures that the notes are always pseudo-random. The last line is to help set-up the ddfs core correctly.



```
do {
    //generate random value
    bit  = ((lfsr >> 0) ^ (lfsr >> 2) ^ (lfsr >> 3) ^ (lfsr >> 5) ) & 1;
    lfsr =  (lfsr >> 1) | (bit << 15);
    tone = lfsr % 3;

    //play tone
    if (tone == 0) { //C
        ddfs_p->set_env(0.1);
        ddfs_p->set_carrier_freq(262);
        sleep_ms(2000);
        ddfs_p->set_env(0.0);
    }
    else if (tone == 1) {//F
            ddfs_p->set_env(0.1);
            ddfs_p->set_carrier_freq(349);
            sleep_ms(2000);
            ddfs_p->set_env(0.0);
    }
    else if (tone == 2) {//B
            ddfs_p->set_env(0.1);
            ddfs_p->set_carrier_freq(494);
            sleep_ms(2000);
            ddfs_p->set_env(0.0);
    }

    //CHOOSE
    sseg->write_1ptn(sseg->h2s(12), 6);  //C
    sseg->write_1ptn(137, 5);            //H
    sseg->write_1ptn(sseg->h2s(0), 3);   //O
    sseg->write_1ptn(sseg->h2s(0), 4);   //O
    sseg->write_1ptn(sseg->h2s(5), 2);   //S
    sseg->write_1ptn(sseg->h2s(14), 1);  //E

    sleep_ms(5000);

    for (int i = 0; i < 8; i++) {
       sseg->write_1ptn(0xff, i);
    }
```

Figure 5: Second Part of the Code

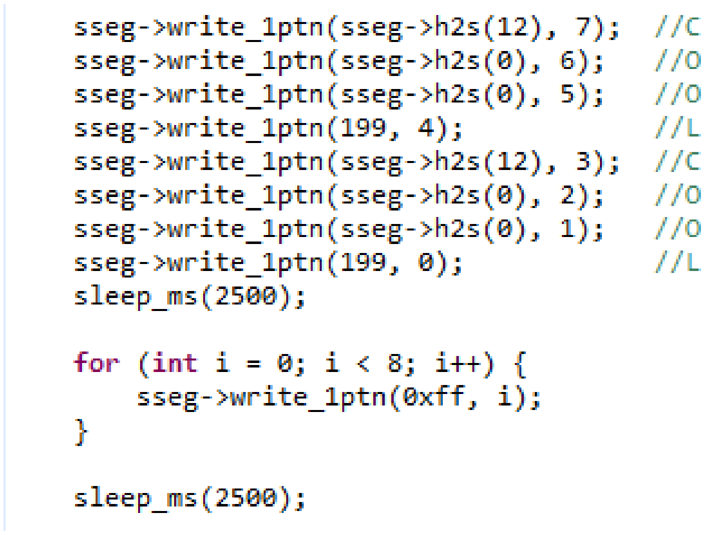
This section shows the start of the do-while loop that is necessary for keeping track of how many correct notes the player has guessed. The first section of the do-while is for random value generation using a linear feedback shift register. The random value can either be 0, 1, or 2, which is associated with C, F, or B. The next section plays a frequency through the speaker using the ddfs core. The frequency played depends on the tone integer, and matches the generated note (if tone is 0, a 262 Hz sound wave is played because that is the frequency of a C4 note). Lastly, the word ”choose” is displayed on the seven-segment display via Chu’s sseg core, prompting the player to press a button on the keyboard. After 5 seconds, the seven-segment display turns off.



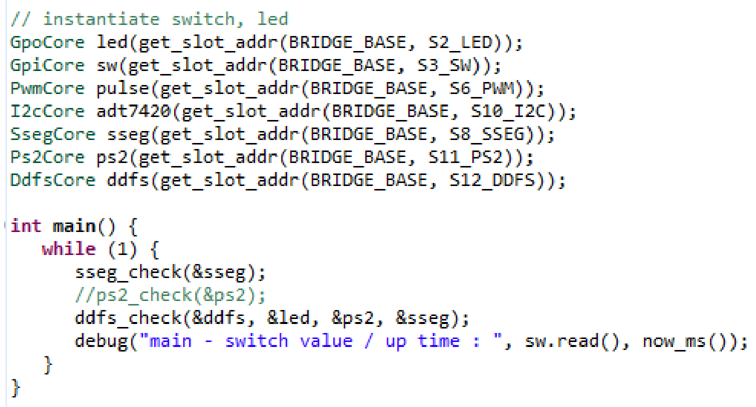
```
        //get mouse guess
        while (!keyboardPress) {
            if (ps2_p->get_kb_ch(&ch)) {
                if (ch == 'c') {
                    guess = 0;
                    keyboardPress = 1;
                }
                else if (ch == 'f') {
                    guess = 1;
                    keyboardPress = 1;
                }
                else if (ch == 'b') {
                    guess = 2;
                    keyboardPress = 1;
                }
                else {
                    keyboardPress = 0;
                }
            }
        }

        if (guess == tone) {
            ++correct;
            sseg->write_1ptn(sseg->h2s(12), 5);  //C
            sseg->write_1ptn(sseg->h2s(0), 3);   //O
            sseg->write_1ptn(sseg->h2s(0), 4);   //O
            sseg->write_1ptn(199, 2);            //L

            sleep_ms(2500);

            for (int i = 0; i < 8; i++) {
                sseg->write_1ptn(0xff, i);
            }
        }
        else {
            correct = 0;
            sseg->write_1ptn(134, 6);  //E
            sseg->write_1ptn(175, 5);   //r
            sseg->write_1ptn(175, 4);   //r
            sseg->write_1ptn(163, 3);  //o
            sseg->write_1ptn(175, 2);   //r

            sleep_ms(2500);

            for (int i = 0; i < 8; i++) {
                sseg->write_1ptn(0xff, i);
            }
        }

        keyboardPress = 0;

    } while (correct < 5);
```

Figure 6: Third Part of the Code

In this section, there is a while-loop that stalls the program while waiting for the keyboard press from the player. Then, using the ps2 core, that keyboard press is recorded, and the guess variable is updated accordingly. This guess variable is then compared with the tone variable. If they match, the correct varibale is incremented, and the word ”cool” is shown on the seven-segment display to let the player know they were correct. If the player was wrong, the correct variable is set back to 0, even if the player had 4 correct guesses previously. The word ”error” is shown on the seven-segment display to let the player know they were wrong. Lastly, the end of the do-while loop can be seen, showing that five correct guesses are needed in order to break out of the loop and win the game.



```
    sseg->write_1ptn(sseg->h2s(12), 7);  //C
    sseg->write_1ptn(sseg->h2s(0), 6);   //O
    sseg->write_1ptn(sseg->h2s(0), 5);   //O
    sseg->write_1ptn(199, 4);            //L
    sseg->write_1ptn(sseg->h2s(12), 3);  //C
    sseg->write_1ptn(sseg->h2s(0), 2);   //O
    sseg->write_1ptn(sseg->h2s(0), 1);   //O
    sseg->write_1ptn(199, 0);            //L
    sleep_ms(2500);

    for (int i = 0; i < 8; i++) {
        sseg->write_1ptn(0xff, i);
    }

    sleep_ms(2500);
```



```
// instantiate switch, led
GpoCore led(get_slot_addr(BRIDGE_BASE, S2_LED));
GpiCore sw(get_slot_addr(BRIDGE_BASE, S3_SW));
PwmCore pulse(get_slot_addr(BRIDGE_BASE, S6_PWM));
I2cCore adt7420(get_slot_addr(BRIDGE_BASE, S10_I2C));
SsegCore sseg(get_slot_addr(BRIDGE_BASE, S8_SSEG));
Ps2Core ps2(get_slot_addr(BRIDGE_BASE, S11_PS2));
DdfsCore ddfs(get_slot_addr(BRIDGE_BASE, S12_DDFS));

int main() {
   while (1) {
      sseg_check(&sseg);
      //ps2_check(&ps2);
      ddfs_check(&ddfs, &led, &ps2, &sseg);
      debug("main - switch value / up time : ", sw.read(), now_ms());
   }
}
```

Figure 7: Last Part of the Code

Once the player has gotten five correct guesses, a positive message is shown on the seven-segment display, letting them know they've won. The program's main function automatically loops, so as soon as the player wins, the game will start back at the beginning, encouraging intensive ear training for the player!

Results

The project went very smoothly overall. Chu's cores allow for easy implementation between the Nexys4 board, keyboard, and speaker. It took some time to iron out the do-while loop logic to make sure that the game worked consistently over a number of trials, but overall, there was nothing too crazy that came up. The final product is a fun game that can easily be scaled to include all the notes on the scale for the advanced musicians out there!