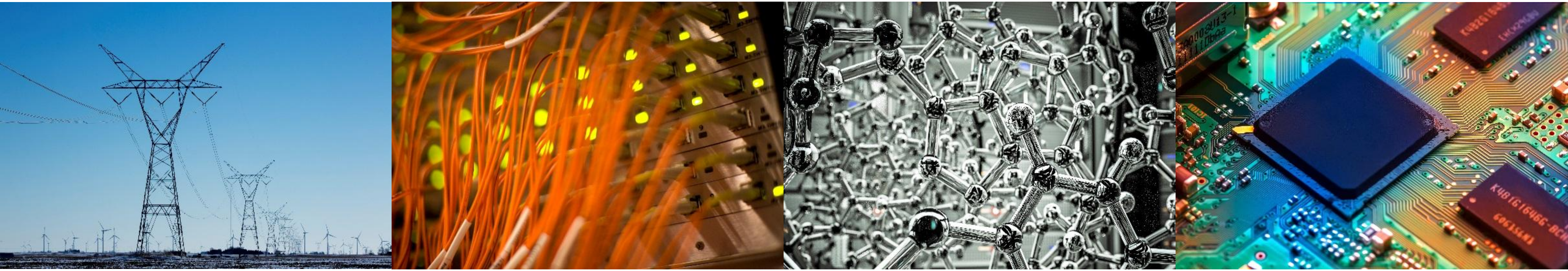
Group 64 - Virtual Piano

ECE 445 Spring 2018

Jeongsub Lee
Zhi Lu



ILLINOIS
Electrical & Computer Engineering
COLLEGE OF ENGINEERING

Introduction

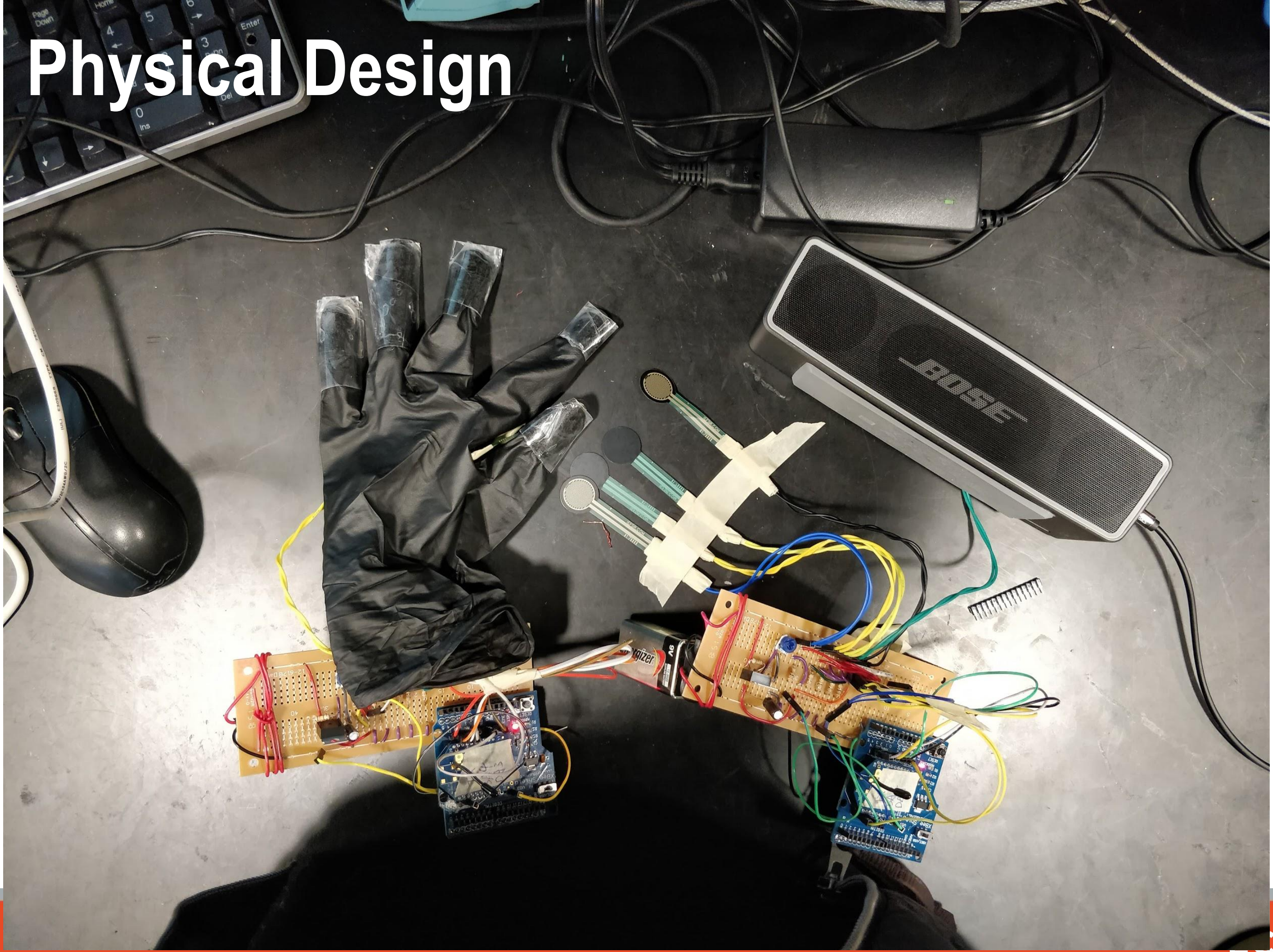
- A convenient way to play piano at any place
- Lower cost piano that could be delivered to broader range of user.
- Replicate features of conventional grand piano

Objective

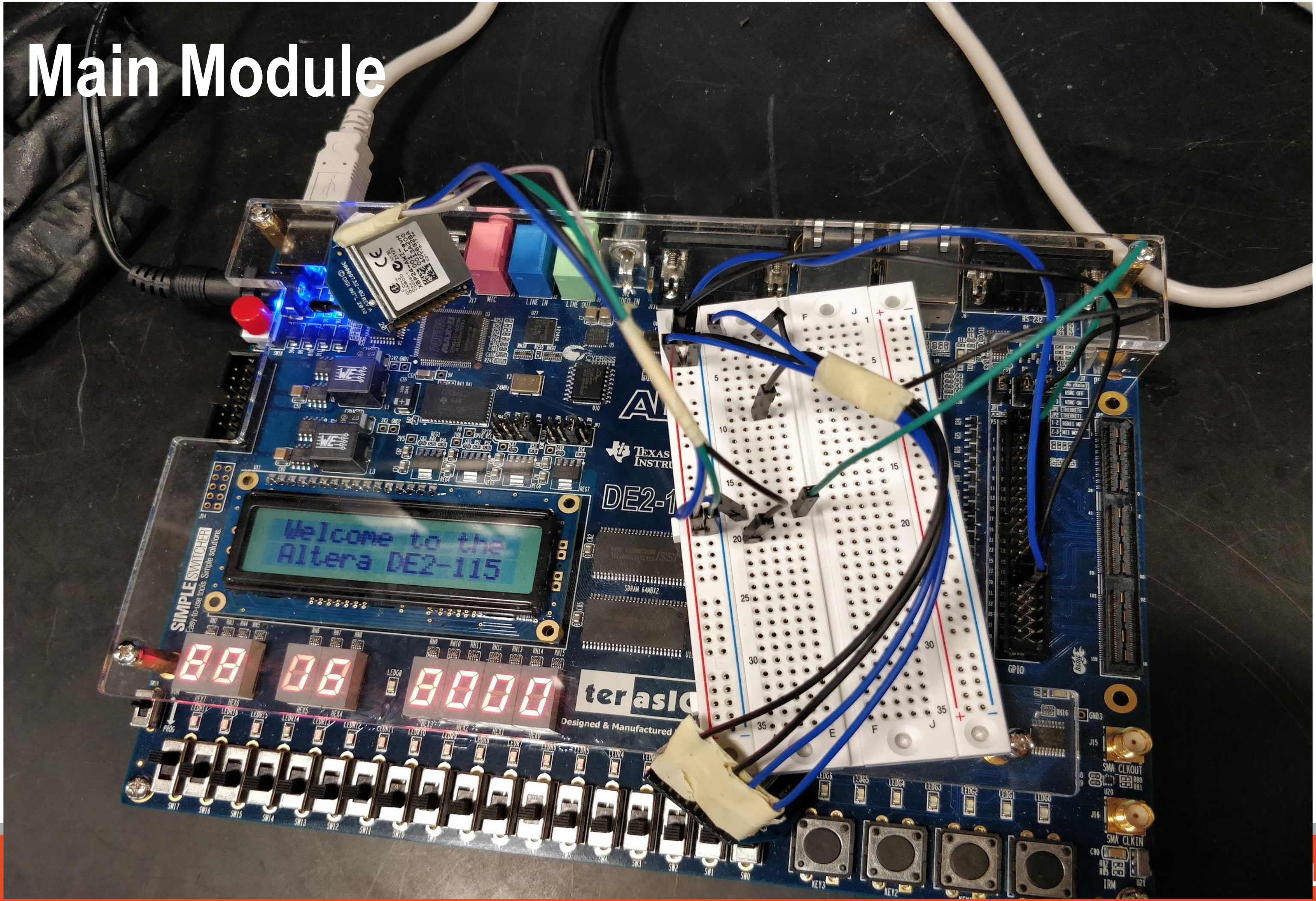
Wanted to incorporate:

- Ability to play different tones
- Ability to mix multiple tones
- Ability to vary volume based on finger pressure
- Different audio filters and effects

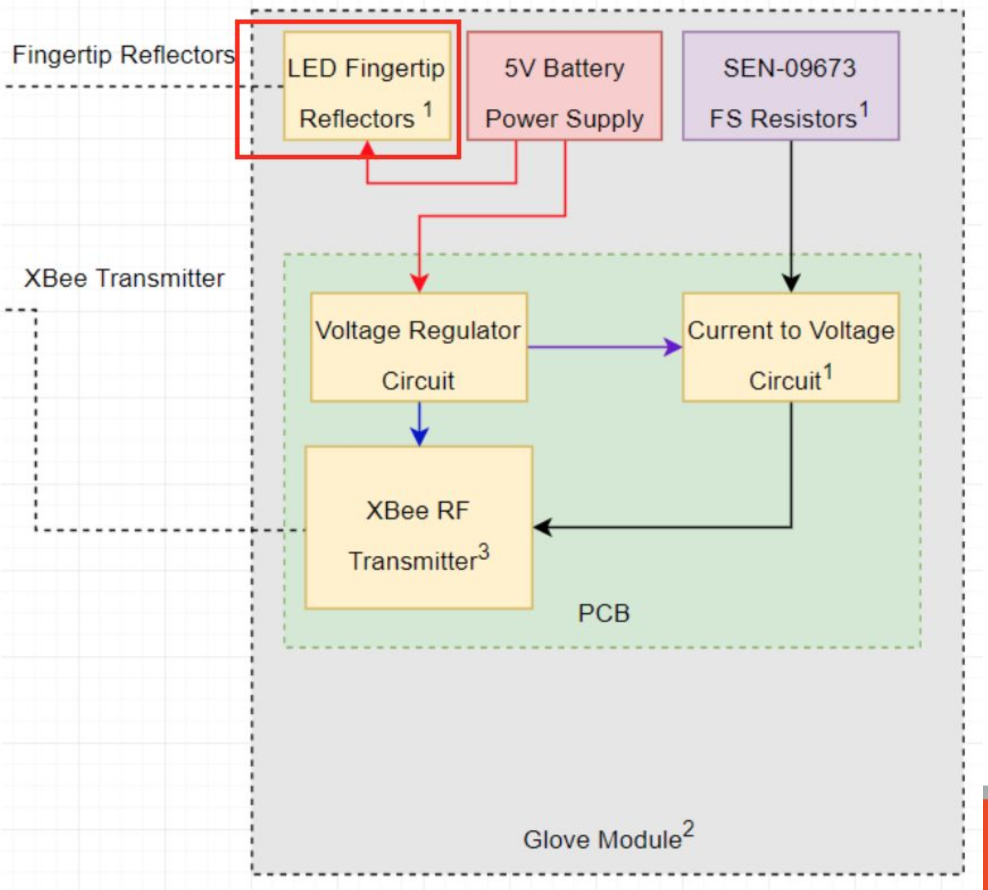
Physical Design



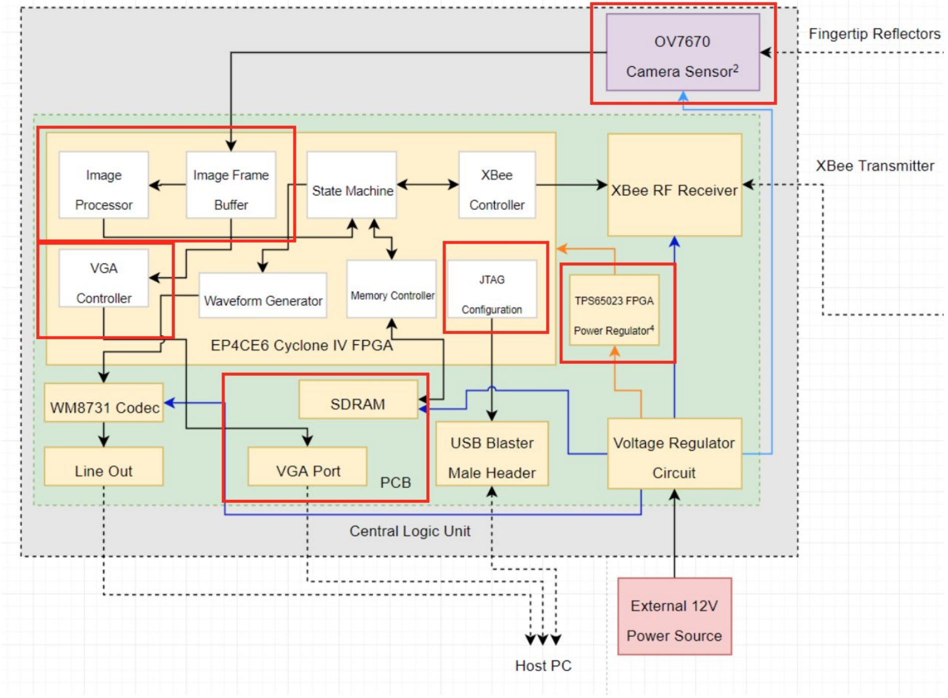
Main Module


Welcome to the
Altera DE2-115
SDRAM 64MBX2
GPIO
SMA CLKOUT
SMA CLKIN
Designed & Manufactured

Glove Module Block Diagram



Main Module Block Diagram



Notes:

1. 5 units in each module for each finger.
2. 2 units for each hand.
3. Using provided ADC in XBee module.
4. Cyclone IV requires multiple input voltages.

Color Key:

- High Level Modules
- PCB Board
- Sensors
- Physical Electronic Modules
- FPGA modules

Arrow Key:

- 1.7 V Power Supply
- 3.3 V Power Supply
- 4 V Power Supply
- 5 V Power Supply
- 6 V Power Supply
- Wired Signal
- Wireless/Abstract Signal

Force Resistive Sensor - Volume control

-Input: Finger Pressure(Analog signal)

Analog to Digital signal Converter in XBEE Pro

-Output: 0x0000 - 0x3FFF(Digital Signal)

Single increment in hex value corresponds to ~1g in pressure as the entire range covers up to 10kg of pressure.

Calculations

- Regulated Voltage Output:
- XBee = 3.3V
- Cyclone IV EP4CE6

  -VCCINT: 1.2V

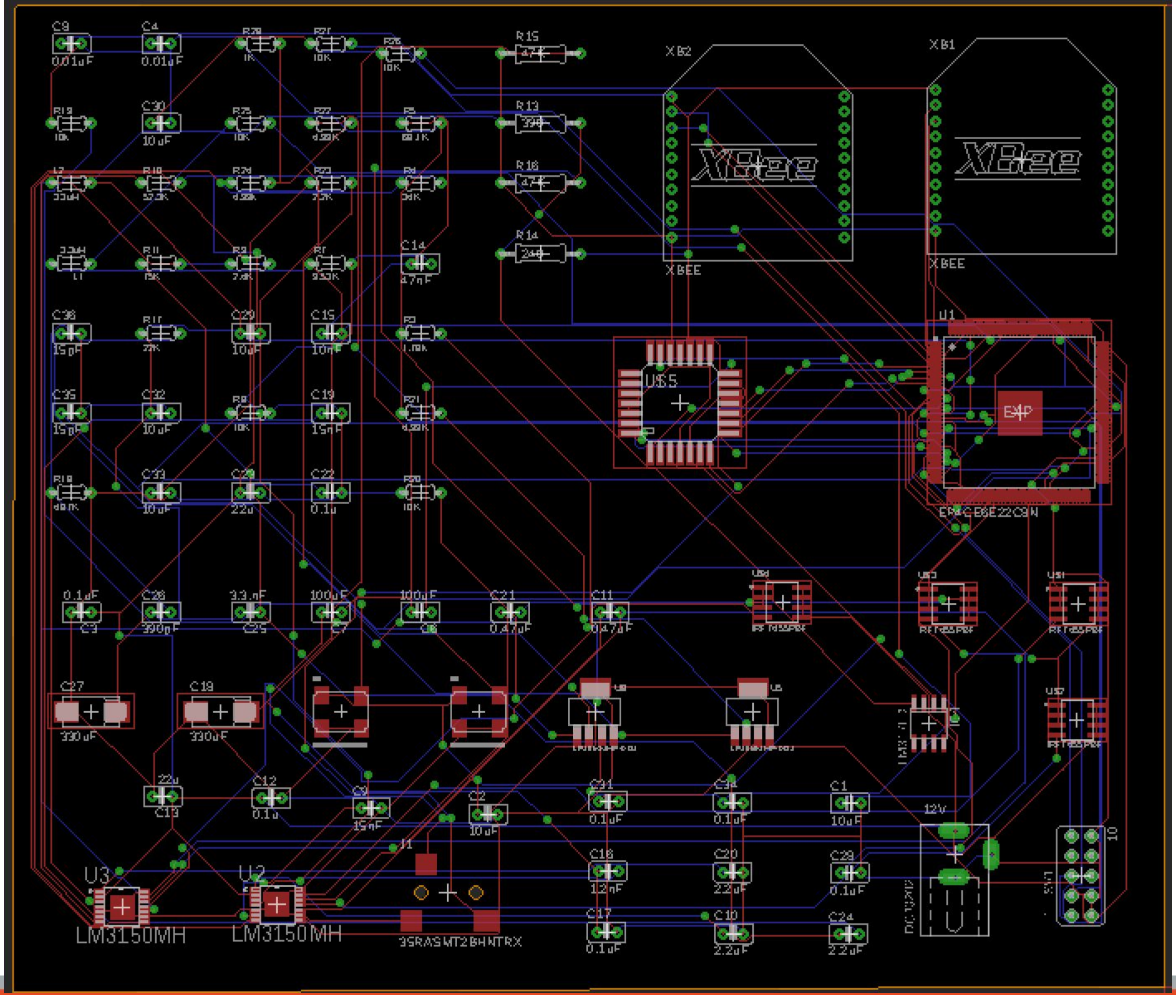
  -VCC_CLKIN:2.5V

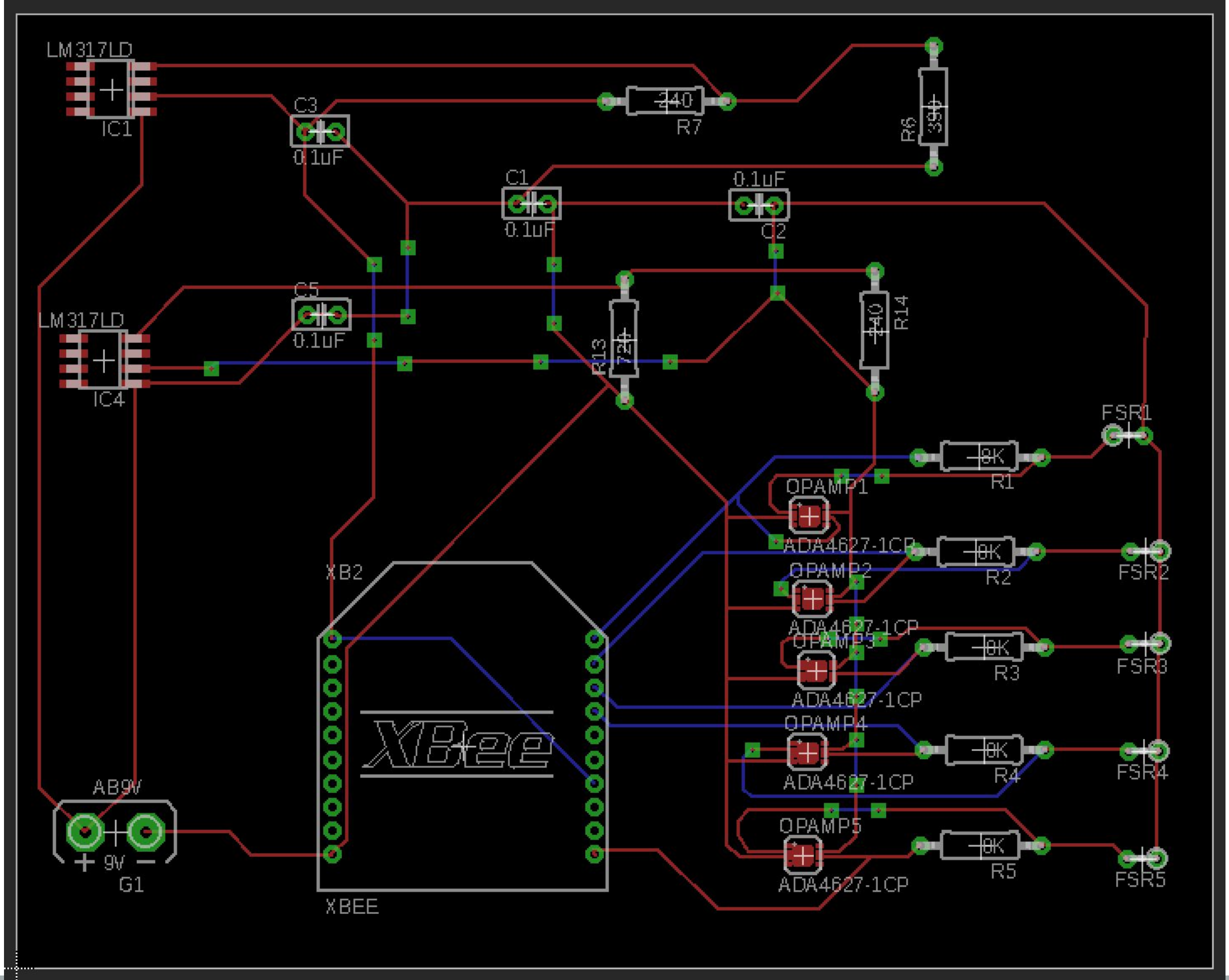
  -VCCIO: 1.8V, 3.3V

  -VCCD_PLL: 1.2V

  -VCCA: 2.5V

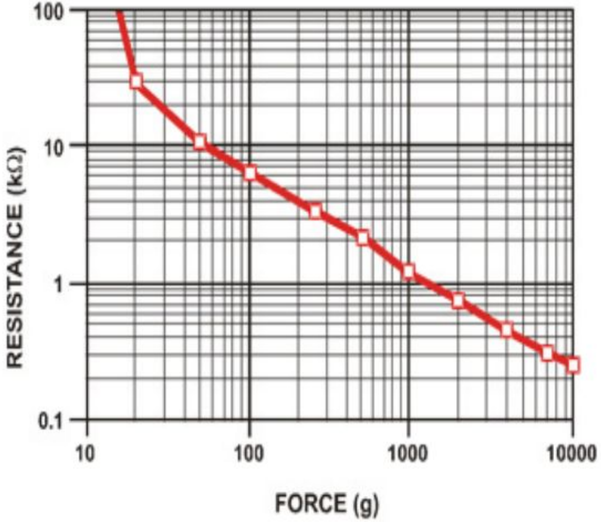
-


XB2
XB1
XBee
XBee
XBEE
XBEE
U1
U$5
LM3150MH
LM3150MH
U3
U2
12V


LM317LD
IC1
C3
0.1uF
R7
R6
C1
0.1uF
0.1uF
C2
C5
0.1uF
LM317LD
IC4
R13
R14
FSR1
R1
OPAMP1
ADA4627-1CP
R2
OPAMP2
ADA4627-1CP
OPAMP3
R3
ADA4627-1CP
OPAMP4
R4
ADA4627-1CP
OPAMP5
R5
ADA4627-1CP
FSR2
FSR3
FSR4
FSR5
XB2
XBee
XBEE
AB9V
9V
G1

Calculations

- Force Sensitive Resistor Resolution:



Rreference = 20K Ohms

Vout = Vin * (Rref/Rtotal)

Data R: 0x0000~0x3FFF

Used FSR range :0.6K ~ 600K

Best Resolution at 0.6*(600/0.6)^½

20K Ohms

FPGA Requirements & Verifications

- Should be able to receive pressure sensor data from the XBee receiver at a rate of 16 Kbps for required resolution.
- Waveform generator must be able to provide 16 bit digital audio samples generated at a PWM frequency to codec to output the required analog waveform.

WM8731 Audio Codec Requirements & Verifications

- Should be able to receive digital samples at a modulated frequency from the FPGA waveform generator and produce a corresponding analog signal.
- Should be able to process input digital samples at a high frequency required for frequency response of piano notes.

XBee Modules Requirements & Verifications

- Transmit data within at least 2 meters reliably.
- Convert the analog data from the pressure sensors (FSR 400) to digital using on-board ADCs.
- Transmit data at 80 Kbps for the processing of pressure sensor readings

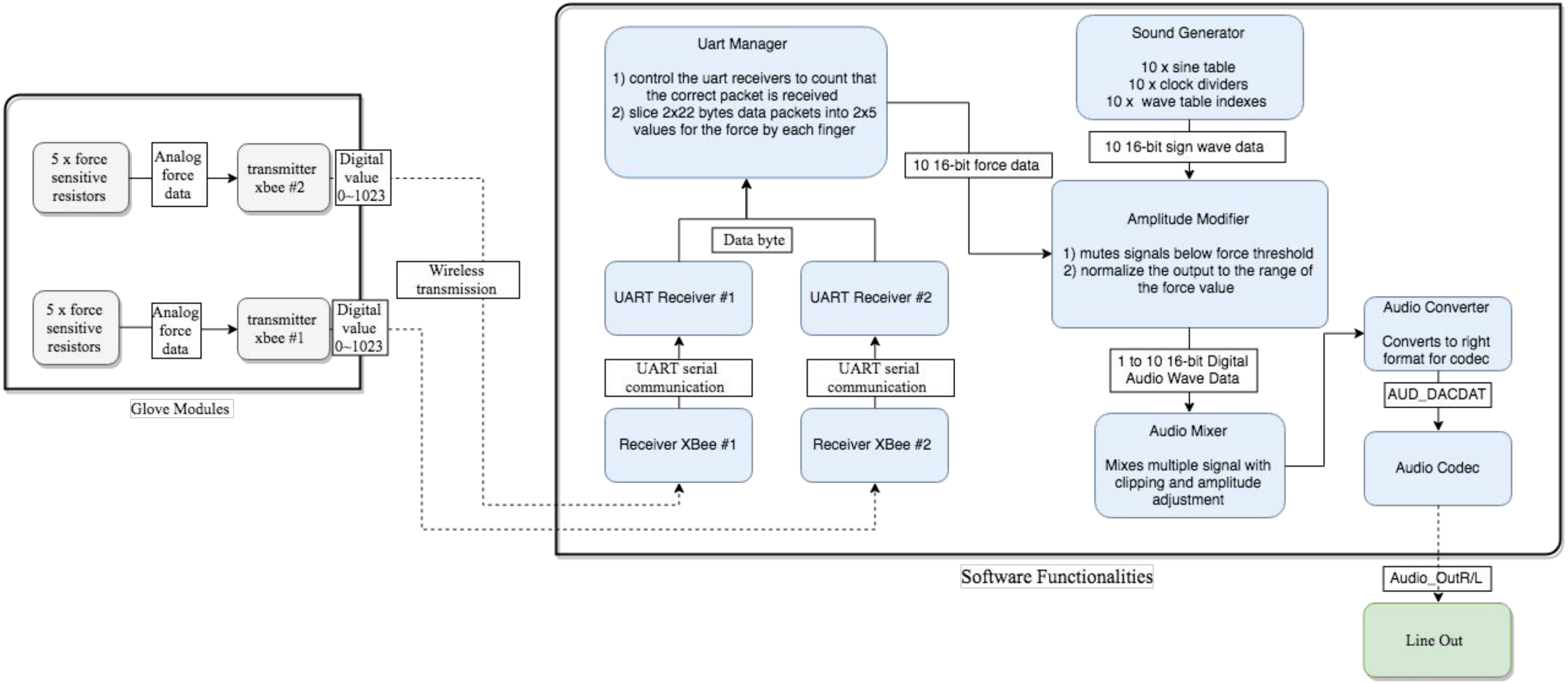
Force Sensitive Resistor R & V

- XBee module has range of analog input that won't be cut off. The Maximum is nearly the VCC+0.3V of the XBEE module which is going to be 3.6V in our design. So, resulting voltage output from the op-amp should not exceed 3.6V for proper voltage mapping.

- Force Sensitive Resistor module needs to be powered by battery. The battery should be able to supply 3.3V to Xbee module and FSR. 9V battery and a subcircuit of voltage regulator is used.

- For proper voltage mapping and greater sensitivity, we need to have linear response, with precise resistance value to achieve maximum resolution. We used a voltage regulator circuit with appropriate choice of resistance.

General Data Flow



Calculations

- Latency (Time from finger pressing to the sound being generated): ~ 7ms
  - Baud rate: 115200 bps
  - XBee transmission speed: 250 kbps
  - Sampling Rate: 1 kHz
  - Size of data packet: 22 bytes
  - Sound generator modules takes the average of 5 force reading samples (5 packets) for output generation
  - (22 bytes * 8 bits/byte * 5 packets) / 115200 bps ~ 7.64ms

Uart Manager Module

```
module Uart_Manager(
    input                CLOCK_50,
    input                GPIO_1,
    input                GPIO_2,

    output [15:0]    o_FSR_data_1,
    output [15:0]    o_FSR_data_2,
    output [15:0]    o_FSR_data_3,
    output [15:0]    o_FSR_data_4,
    output [15:0]    o_FSR_data_5,
    output [15:0]    o_FSR_data_6,
    output [15:0]    o_FSR_data_7,
    output [15:0]    o_FSR_data_8,
    output [15:0]    o_FSR_data_9,
    output [15:0]    o_FSR_data_10,


    //DEBUGGING
    output received_data,
    output [31:0]    DEBUG_total_bytes_count,
    output [7:0]     DEBUG_received_byte_1,DEBUG_received_byte_2,
    output           DEBUG_RX_DV_1,
    output [7:0]     DEBUG_Clock_Count,
    output [4:0]     DEBUG_byte_counter_1, DEBUG_byte_counter_2,
    output [2:0]     DEBUG_SM_Main
);
```

```
/*=================== Receiver Submodules ==============*/
// CLK per bit = Clock Freq/Baud = 50000000/115200 = 434
UART_RX #(434) uart_1 (
    .i_Clock(CLOCK_50),
    .i_RX_Serial(GPIO_1),
    .o_RX_DV(RX_DV_1),
    .o_RX_Byte(received_byte_1)
    //,.DEBUG_Clock_Count(DEBUG_Clock_Count)
    //,.DEBUG_SM_Main(DEBUG_SM_Main)
);

UART_RX #(434) uart_2 (
    .i_Clock(CLOCK_50),
    .i_RX_Serial(GPIO_2),
    .o_RX_DV(RX_DV_2),
    .o_RX_Byte(received_byte_2)
    ,.DEBUG_SM_Main(DEBUG_SM_Main)
);
```

Uart Manager Cont.

```
always @(posedge CLOCK_50) begin
    if (RX_DV_1 == 1) begin
        // set the FSR data depending on the byte being received
        case (byte_counter_1)
        8'd11   : FSR_temp_1[15:8]      <= received_byte_1;
        8'd12 : FSR_temp_1[7:0]         <= received_byte_1;
        8'd13 : FSR_temp_2[15:8]        <= received_byte_1;
        8'd14 : FSR_temp_2[7:0]         <= received_byte_1;
        8'd15 : FSR_temp_3[15:8]        <= received_byte_1;
        8'd16 : FSR_temp_3[7:0]         <= received_byte_1;
        8'd17 : FSR_temp_4[15:8]        <= received_byte_1;
        8'd18 : FSR_temp_4[7:0]         <= received_byte_1;
        8'd19 : FSR_temp_5[15:8]        <= received_byte_1;
        8'd20 :
        begin
            FSR_temp_5[7:0]     <= received_byte_1;
              reg [15:0] FSR_temp_5   _counter_1+1;

                               gathered, take the average then clear
            if (sample_counter_1 >= 5) begin
                FSR_1 <= FSR_sum_1/sample_counter_1;
                FSR_2 <= FSR_sum_2/sample_counter_1;
                FSR_3 <= FSR_sum_3/sample_counter_1;
                FSR_4 <= FSR_sum_4/sample_counter_1;
                FSR_5 <= FSR_sum_5/sample_counter_1;

                FSR_sum_1 <= 0;
                FSR_sum_2 <= 0;
                FSR_sum_3 <= 0;
                FSR_sum_4 <= 0;
                FSR_sum_5 <= 0;
                sample_counter_1 = 0;
            end
            else begin
                FSR_sum_1 <= FSR_sum_1 + FSR_temp_1;
                FSR_sum_2 <= FSR_sum_2 + FSR_temp_2;
                FSR_sum_3 <= FSR_sum_3 + FSR_temp_3;
                FSR_sum_4 <= FSR_sum_4 + FSR_temp_4;
                FSR_sum_5 <= FSR_sum_5 + FSR_temp_5;
            end

        end
        default : begin end
        endcase


    end
end
```

Sound Generator

```
module sound_generator (
    input           CLOCK_50,

    input [15:0]    i_FSR_data_1,
    input [15:0]    i_FSR_data_2,
    input [15:0]    i_FSR_data_3,
    input [15:0]    i_FSR_data_4,
    input [15:0]    i_FSR_data_5,
    input [15:0]    i_FSR_data_6,
    input [15:0]    i_FSR_data_7,
    input [15:0]    i_FSR_data_8,
    input [15:0]    i_FSR_data_9,
    input [15:0]    i_FSR_data_10,

    output [15:0]   sound_gen_out
);
```

```
/*=============== Helper Functions ===============*/
task divide_clock;
inout reg[28:0] divider;
inout reg[7:0] index;
input reg[28:0] divide_param;
begin
    if (divider == 0)
    begin
        divider <= divide_param;
        index <= index + 1;
        if (index >= WAVE_PERIOD)            // wave table period
            index <= 0;
    end
    else divider <= divider -1;
end
endtask
```

```
sine_table sine_table_inst_1(
   .index(table_index_1),
   .signal(sine_wave_out_1)
);
```

```
Amplitude_Modifier  #(
    .FORCE_MAX(1023),            // (theore
    .FORCE_MIN(0),                 // (t
    .THRESHOLD(80)               // take a
) amp_mod_1(
    .CLOCK_50(CLOCK_50),
    .wave_data(sine_wave_out_1),
    .force_data(i_FSR_data_1),
    .o_amp_mod_wave(amp_mod_wave_1)
);
```

Sound Generator Cont.

```
/*================== Audio Mixer mo
Wave_Mixer wav_mixer (
    .CLOCK_50(CLOCK_50),
    .i_wave_1(sine_wave_out_1),
    .i_wave_2(sine_wave_out_2),
    .i_wave_3(sine_wave_out_3),
    .o_mixed_wave(mixed_wave)
);

wire[15:0] mixed_wave;
```

```
/*=========== Constants Declarations ================*/
localparam WAVE_PERIOD = 255;                          // need t
localparam CLOCK_FREQ = 50_000_000-340_000;            // 50 Mhz clo

// List of note frequencies
localparam NOTE_1 = 261.626;                 // C4 (Middle C)
localparam NOTE_2 = 293.665;                 // D4
localparam NOTE_3 = 329.628;                 // E4
localparam NOTE_4 = 349.228;                 // F4
localparam NOTE_5 = 391.995;                 // G4
localparam NOTE_6 = 440.0;                   // A4
localparam NOTE_7 = 493.883;                 // B4
localparam NOTE_8 = 523.251;                 // C5
localparam NOTE_9 = 587.330;                 // D5
localparam NOTE_10 = 659.255;              // E5
/* References:
```

```
always @(posedge CLOCK_50) begin
    divide_clock(clk_divider_1, table_index_1, CLOCK_FREQ/(NOTE_1 * WAVE_PERIOD));
    div            _2, table_index_2, CLOCK_FREQ/(NOTE_2 * WAVE_PERIOD));
    div            _3, table_index_3, CLOCK_FREQ/(NOTE_3 * WAVE_PERIOD));
    div            _4, table_index_4, CLOCK_FREQ/(NOTE_4 * WAVE_PERIOD));
    div            _5, table_index_5, CLOCK_FREQ/(NOTE_5 * WAVE_PERIOD));
    divide_clock(clk_divider_6, table_index_6, CLOCK_FREQ/(NOTE_6 * WAVE_PERIOD));
    divide_clock(clk_divider_7, table_index_7, CLOCK_FREQ/(NOTE_7 * WAVE_PERIOD));
    divide_clock(clk_divider_8, table_index_8, CLOCK_FREQ/(NOTE_8 * WAVE_PERIOD));
    divide_clock(clk_divider_9, table_index_9, CLOCK_FREQ/(NOTE_9 * WAVE_PERIOD));
    divide_clock(clk_divider_10, table_index_10, CLOCK_FREQ/(NOTE_10 * WAVE_PERIOD));

end
```

Definition:
sound_generator.v:52

Conclusion & Further Work

Future Work:

- Make use of Electromagnetic Force or spring and replicate piano-touch.
- Virtual mapping of keyboard using webcams.

Feel Free to Ask Question!