BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Course No.: EEE 304

Course Title: Digital Logic Design Lab

Project Report

Project Title:

FPGA-BASED FREQUENCY SPECTRUM ANALYZER

Submitted to:

Dr. S. M. Mahbubur Rahman
Professor,
Department of EEE,
BUET

&

Shuvro Chowdhury
Lecturer,
Department of EEE,
BUET

Group No: 06

Level: 3 Term: I
Section: A1
Department: EEE

Date of Submission : 27.05.2015

Group Members:-
Student No.:

(i) 1106028
(ii) 1106029
(iii) 1106030
(iv) 1106031
(v) 1106032
(vi) 1106033

Project Title:

Design and implementation of an FPGA-based real-time frequency spectrum analyzer using Verilog HDL and FFT algorithm.

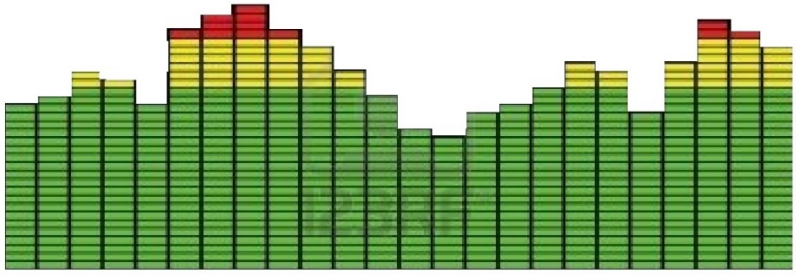
Project Planning:

Objective:

Our objective is to build a frequency spectrum analyzer by implementing an FPGA-based processor which will perform the complex FFT algorithm. This project should demonstrate how to implement a complicated algorithm using Verilog HDL and FPGA.

Spectrum Analyzer:

A **spectrum analyzer** measures the magnitude of an input signal versus frequency within the full frequency range of the instrument. In this project we will take an analog signal, typically audio signal as input, convert it into digital signal, then use discrete signal processing to analyze the data and show the power vs. frequency visually in a display.



Spectrum Analyzer

Flow diagram:

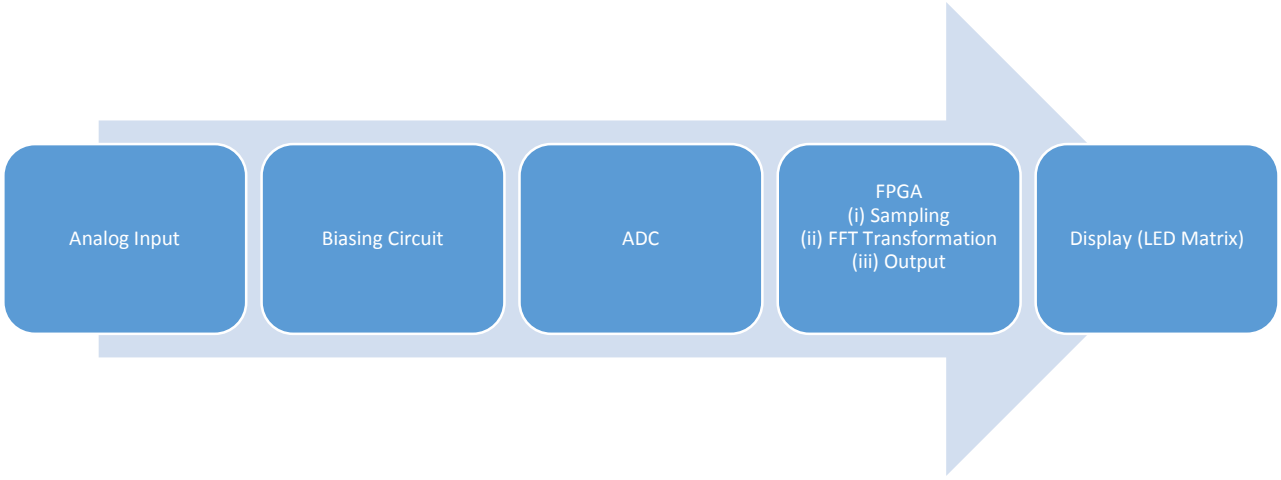
(i) An analog input signal, typically audio signal from a microphone or some other source is taken.

(ii) The AC signal is coupled with a DC biasing voltage.

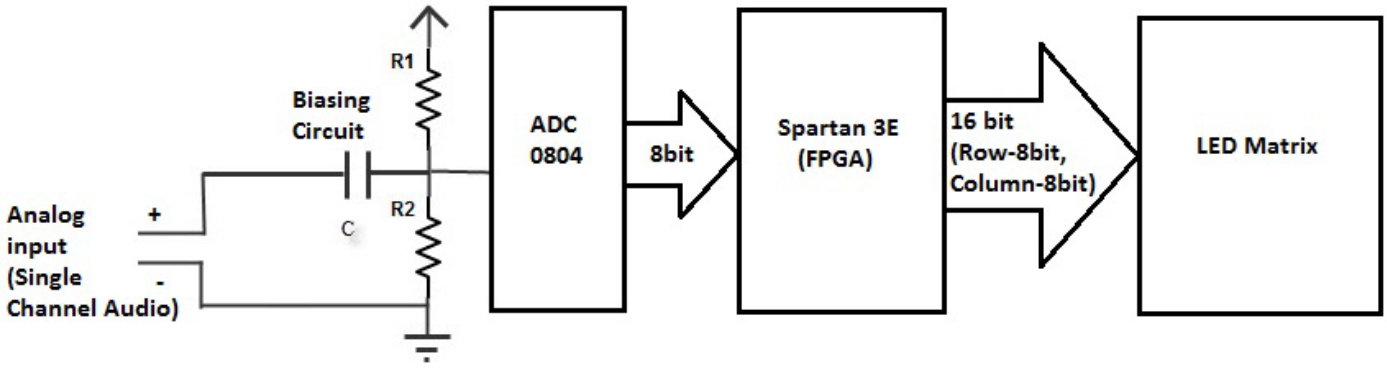
(iii) The amplified signal is fed into an ADC which will convert it into digital signal.

(iv) The digital signal will is processed by the logic circuitry of a FPGA which will
   a. Sample the signal and store the time domain data
   b. Convert the time domain signal into frequency domain data via FFT algorithm
   c. The frequency domain data will be used to visualize the spectrum by the output circuitry which will drive the LED matrix.

(v) Magnitude of different bands of frequency is represented by heights of columns of lights in the LED Matrix.



Circuit Diagram:



Input circuit and biasing:

A single channel from a dual channel audio source is taken via 3.5 mm audio jack. The analog ac signal is coupled to a 2.5 V DC value via a coupling capacitor. This value is fed into the ADC 0804.

ADC:

The ADC 0804 converts the analog input into 8 bit digital output. This 8 bit digital signal works as the input for the Spartan 3E FPGA.

FPGA:

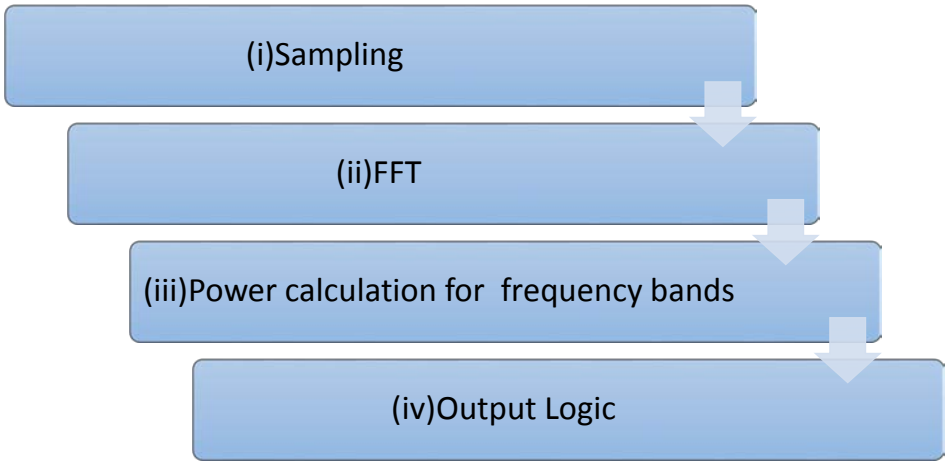
The FPGA stores the digital input data, performs discrete time signal processing. The frequency domain data is used to compute power in 8 frequency bands in logarithmic scales. This data is used to drive an LED Matrix.

LED Matrix:

8 rows and 8 columns of the LED Matrix is driven by 16 bit output from the FPGA. Different columns of the LED matrix are turned one at a time. This is done at a high frequency so that the human eye cannot realize the time when they are off.

Algorithm and Flowchart:

Flowchart:



(i)Sampling:

The first step in discrete time signal processing is sampling the continuous time signal. Two important points regarding sampling:

(a)The sampling frequency:

The sampling frequency limits the highest frequency that can be interpreted in the frequency domain. A signal sample at frequency $2f$ can contain a highest frequency of f.

(b)The number of samples processed:

The number of samples processed by Discrete Fourier Transformation limits the resolution of the frequency domain data. Processing $2n$ samples will result in n frequency domain data plus DC.

(ii)FFT:

The Fast Fourier Transform (FFT) which is an efficient algorithm for calculating the Discrete Fourier Transform is a vital part of signal processing. We used the Decimation In Time Fast Fourier Transform which is the simplest form of the algorithm. It converts n time domain data (the stored samples) into n complex frequency domain data.

Decimation In Time Fast Fourier Transform:

The radix-2 decimation-in-time algorithm rearranges the discrete Fourier transform (DFT) equation into two parts: a sum over the even-numbered discrete-time indices $n=[0,2,4,\ldots,N-2]$ and a sum over the odd-numbered indices $n=[1,3,5,\ldots,N-1]$ as in

$$
\begin{aligned}
X(k) &= \sum_{n=0}^{N-1} x(n) e^{-\left(i\frac{2\pi nk}{N}\right)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\left(i\frac{2\pi\times(2n)k}{N}\right)} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-\left(i\frac{2\pi(2n+1)k}{N}\right)} \\
&= \sum_{n=0}^{\frac{N}{2}-1} x(2n) e^{-\left(i\frac{2\pi nk}{\frac{N}{2}}\right)} + e^{-\left(i\frac{2\pi k}{N}\right)} \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) e^{-\left(i\frac{2\pi nk}{\frac{N}{2}}\right)} \\
&= \mathrm{DFT}_{\frac{N}{2}}[[x(0), x(2), \ldots, x(N-2)]] + W_N^k \mathrm{DFT}_{\frac{N}{2}}[[x(1), x(3), \ldots, x(N-1)]]
\end{aligned}
$$

This method of dividing the indices into even an odd parts is used recursively to achieve the Decimation in time FFT algorithm which includes the stages:

Bit reversed sorting:

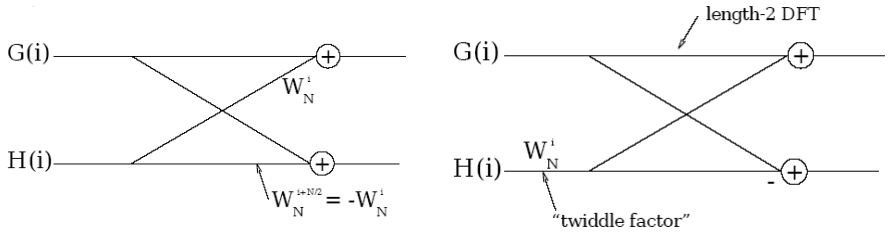
If the input signal data are placed in bit-reversed order before beginning the FFT computations, the outputs of each butterfly throughout the computation can be placed in the same memory locations from which the inputs were fetched, resulting in an in-place algorithm that requires no extra memory to perform the FFT.

Example: N=8

In-order index	In-order index in binary	Bit-reversed binary	Bit-reversed index
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

The butterfly stages:

A basic butterfly operation is shown in Figure, which requires only *N*/2 twiddle-factor multiplies per stage. It is worthwhile to note that, after merging the twiddle factors to a single term on the lower branch, the remaining butterfly is actually a length-2 DFT. The theory of multi-dimensional index maps shows that this must be the case, and that FFTs of any factorable length may consist of successive stages of shorter-length FFTs with twiddle-factor multiplications in between.

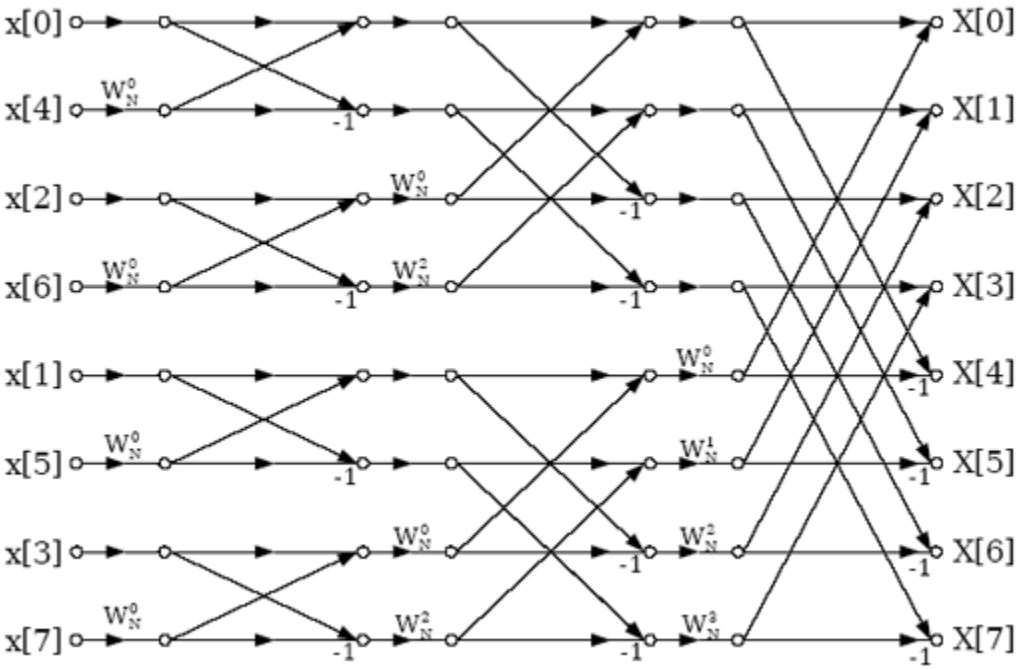


Twiddle Factors:

The twiddle factors $W_N^n = e^{-i\frac{2\pi n}{N}}$ are pre-calculated complex numbers which are stored in an Look Up Table and used in the butterfly stages.

The Complete Algorithm:

The complete algorithm consists of bit reversal sorting of n samples followed by log(n) stages of butterfly. An example for n=8 is shown in the figure:



Output of FFT algorithm:

The output of FFT algorithm is frequency domain data with X[0] representing the DC component, X[1]-X[N/2], the positive frequency magnitude and phases and X[N/2+1]-X[N] the negative frequency magnitude and phases in form of complex numbers.

(iii)Power Calculation:

To calculate the power of a band we take the squares of the magnitude of the frequency domain data corresponding to that band and sum them up. As there are both positive and negative frequencies, we use only the positive frequencies and multiply corresponding magnitude by 2 to compensate for the exclusion of negative frequencies.

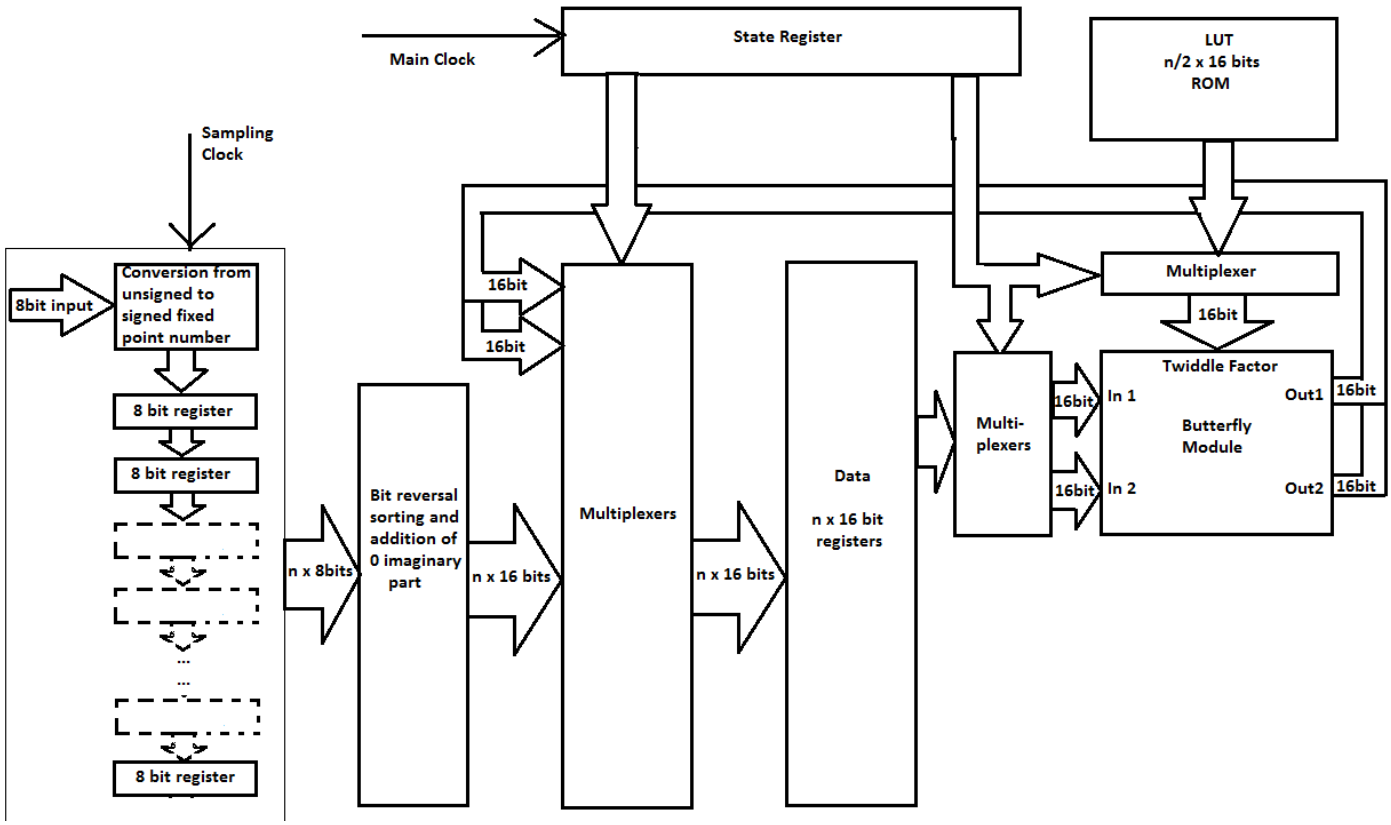
(iv)Output Logic:

The power of each band is represented in logarithmic scale as heights of different columns in the graph.

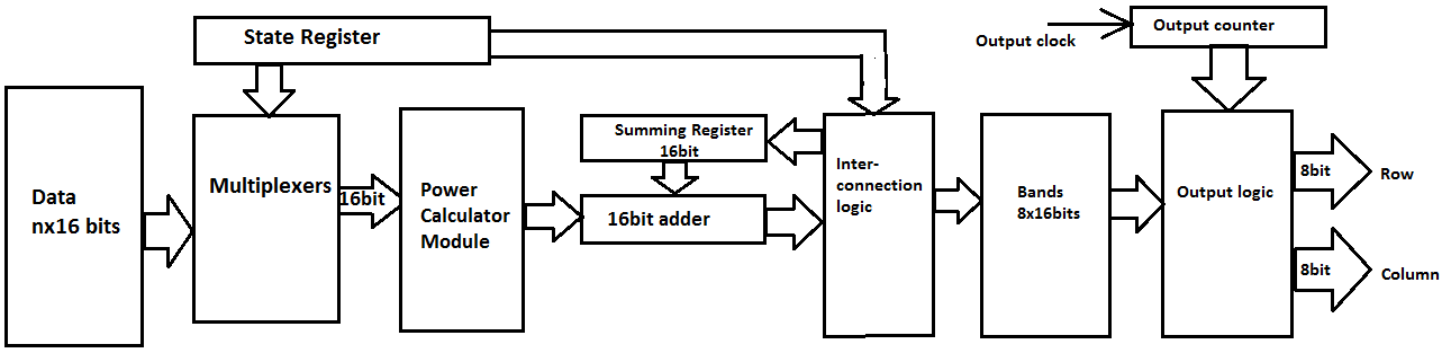
Digital Logic Design: Implementation in FPGA

Block Diagrams:

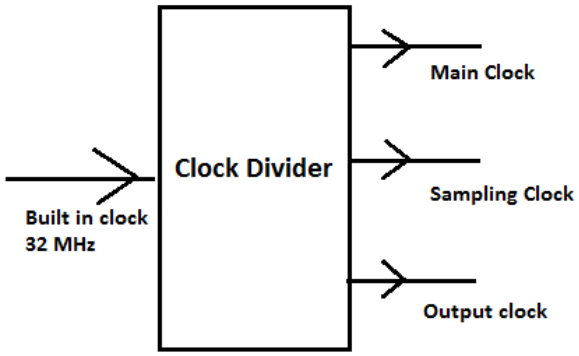
It is difficult to fit the whole circuit into one diagram, so it is divided into parts:



Sampling and FFT units



Power Calculation and output unit



Clock Divider

Number Representation: Fixed point numbers

As all the operations are done on fractional values we used signed 8bit fixed point numbers with the point set before the MSB and a scaling factor of 1/128.

Sign bit	(.)	MSB	… …. …. …. …. ….. ….	LSB

Complex Number Representation:

Complex numbers are stored as 16 bit values with the first 8 bits representing the real part and the later 8 bit representing the imaginary part.

Clock Division:

As the processes of sampling, main calculation and output are done in different frequencies, it is necessary to divide the built in clock into different frequency clocks. We used a simple counter for this purpose with its different bits representing different clocks.

Sampling block:

On every positive edge of the sampling clock, older samples move forward in the queue and new sample is taken as a signed fixed point number.

Verilog Code:

```
reg signed [7:0] samples[n-1:0];

//**********sampling block**************
 always @(posedge sclk)
  begin

   samples[0] <= in-8'd128;
   for (i=1;i<n;i=i+1)
    samples[i]<=samples[i-1];
  end
```

The State Register:

The state register represents the current state of the FSM. It changes at the positive edge of the main clock according to next state logic. We divided the register into two parts for convenience. They are called stage and count.

Verilog Code:

```
//**********states and stages**************
 wire [sts-1:0] stage;
 wire [nl2-2:0] cnt;
 reg [nl2+sts-2:0] state;
 assign cnt = state[nl2-2:0];
 assign stage = state[nl2+sts-2:nl2-1];
```

Bit reversal sorting:

At stage 0 the samples are sorted in reversed bit order and they are stored in the data registers. The imaginary parts are assigned 0 value.

Verilog Code:

```
//**********bit reversal**************
 wire [nl2-1:0] c[n-1:0];
 wire [nl2-1:0] r[n-1:0];
 genvar k,l;
 generate
  for(k=0;k<n;k=k+1)
   begin:loop1
    assign c[k]=k;
    for(l=0;l<nl2;l=l+1)
     begin:loop2
      assign r[k][l]=c[k][nl2-1-l];
     end
   end
 endgenerate
```

```
case (stage)
      0:
       begin
        for(i=0;i<n;i=i+1)
          begin
            data[c[i]]<={samples[r[i]], 8'b0};
          end

        state[nl2+sts-2:nl2-1]<=1;
        state[nl2-2:0]<=0;
       end
```

Data Registers:

N x 16 bit registers which stores the data. Different operations are done on this data.

Verilog Code:

```
reg signed [15:0] data[n-1:0];
```

Butterfly module:

This is the main arithmetic module which performs the butterfly operation on fixed point complex numbers.

Out1 = in1 + in2*tf; out2= in1-in2*tf

Where in1, in2 are input complex numbers, out1, out2 are output 16 bit complex numbers and tf is the twiddle factor.

Verilog Code:

```
module butterfly(in1, in2, tf, out1, out2);

 input [15:0] in1, in2, tf;
 output [15:0] out1, out2;

 wire signed [7:0] in1r, in1i, in2r, in2i, tfr, tfi,out1r, out1i, out2r, out2i;
 wire signed [15:0] i1r, i1i, i2r, i2i, tr, ti, t1r, t1i, t2r, t2i, o1r, o1i, o2r, o2i;

 assign in1r[7:0] = in1[15:8],
  in1i[7:0] = in1[7:0],
  in2r[7:0] = in2[15:8],
  in2i[7:0] = in2[7:0],
  tfr[7:0] = tf[15:8],
  tfi[7:0] = tf[7:0],
  out1[15:8] = out1r[7:0],
  out1[7:0] = out1i[7:0],
  out2[15:8] = out2r[7:0],
  out2[7:0] = out2i[7:0];
```

```
assign i1r=in1r,
   i1i=in1i,
   i2r=in2r,
   i2i=in2i,
   tr=tfr,
   ti=tfi;

assign t1r=i1r<<7,
 t1i=i1i<<7;

assign t2r=i2r*tr-i2i*ti,
 t2i=i2r*ti+i2i*tr;

assign o1r=t1r+t2r,
 o1i=t1i+t2i;

assign o2r=t1r-t2r,
 o2i=t1i-t2i;

assign out1r[7:0]=o1r[15:8],
 out1i[7:0]=o1i[15:8];

assign out2r[7:0]=o2r[15:8],
 out2i[7:0]=o2i[15:8];

endmodule
```

LUT (Look Up Table):

The LUT contains the twiddle factors for the butterfly operation. It contains N/2 complex numbers.

Verilog Code:

```
//**************LUT*******************
 wire [15:0] lut[n/2-1:0];
 assign lut[0] = 16'b01111111_00000000,
        lut[1] = 16'b01111101_00011000,
        lut[2] = 16'b01110110_00110000,
        lut[3] = 16'b01101010_01000111,
        lut[4] = 16'b01011010_01011010,
        lut[5] = 16'b01000111_01101010,
        lut[6] = 16'b00110000_01110110,
        lut[7] = 16'b00011000_01111101,
        lut[8] = 16'b00000000_01111111,
        lut[9] = 16'b11100111_01111101,
        lut[10] = 16'b11001111_01110110,
```

```
lut[11] = 16'b10111000_01101010,
lut[12] = 16'b10100101_01011010,
lut[13] = 16'b10010101_01000111,
lut[14] = 16'b10001001_00110000,
lut[15] = 16'b10000010_00011000;
```

Power Calculation:

This module calculates an 8 bit value which is proportional to the power corresponding to a frequency. If the frequency domain data is *a + ib*,

$$aval \propto (a^2 + b^2)$$

Verilog Code:

```
//*********power value**************
 wire [15:0] val;
 wire signed [7:0] re, im;
 wire signed [15:0] re2, im2, sum;
 wire [7:0] aval;
 wire sc;
 assign sc=(stage==nl2+2);
 assign val = data[cnt];
 assign re[7:0] = sc? data[n/2][15:8]:val[15:8]<<1,
  im[7:0]=sc? data[n/2][7:0]: (val[7:0]<<1),
  re2=re,
  im2=im,
  sum=re2*re2+im2*im2,
  aval=sum[14]?sum[14:6]-1:sum[14:6];
```

Output Logic: Driving the LED Matrix

Different columns of the LED matrix are turned one at a time. This is done at a high frequency so that the human eye cannot realize the time when they are off.

Verilog Code:

```
 //final output
 reg [ofd+2:0] ocount;
 wire [7:0] cur;
 wire [2:0] cc;
 assign cc= ocount[ofd+2:ofd],
  cur = outr[cc],
  column = 8'b10_00_00_00>>cc;
 assign row[7]=~cur[7];
 genvar m;
 generate
  for(m=0;m<7;m=m+1)
   begin:loop
    assign row[m]=row[m+1]&(~cur[m]);
   end
 endgenerate
```

Complete Verilog Code:

in is the input 8 bit data from the ADC, ***clkin*** is the built in 32 MHz clock, ***column*** and ***row*** are each 8 bit outputs for driving the LED matrix:

```
module main(in, row, column, clkin);
 //**********parmeters**************
 parameter n=32;
 parameter nl2=5;
 parameter cfd=4;
 parameter sfd = 17;
 parameter ofd = 17;
 parameter sts= 4;

 //**********inputs and outputs**************
 input [7:0] in;
 output [7:0] row, column;
 input clkin;
 wire clk, sclk;

 //**********states and stages**************
 wire [sts-1:0] stage;
 wire [nl2-2:0] cnt;
 reg [nl2+sts-2:0] state;
 assign cnt = state[nl2-2:0];
 assign stage = state[nl2+sts-2:nl2-1];


 //**********samples and data**************
 reg signed [7:0] samples[n-1:0];
 reg signed [15:0] data[n-1:0];
 integer i;

 //**************LUT********************
 wire [15:0] lut[n/2-1:0];
 assign lut[0] = 16'b01111111_00000000,
        lut[1] = 16'b01111101_00011000,
        lut[2] = 16'b01110110_00110000,
        lut[3] = 16'b01101010_01000111,
        lut[4] = 16'b01011010_01011010,
        lut[5] = 16'b01000111_01101010,
        lut[6] = 16'b00110000_01110110,
        lut[7] = 16'b00011000_01111101,
        lut[8] = 16'b00000000_01111111,
        lut[9] = 16'b11100111_01111101,
```

```
        lut[10] = 16'b11001111_01110110,
        lut[11] = 16'b10111000_01101010,
        lut[12] = 16'b10100101_01011010,
        lut[13] = 16'b10010101_01000111,
        lut[14] = 16'b10001001_00110000,
        lut[15] = 16'b10000010_00011000;
```

```
//**********butterfly**************
wire [15:0] bfin1, bfin2, tf, bfout1, bfout2;
butterfly bfmod (bfin1, bfin2, tf, bfout1, bfout2);

//**********butterfly indice**************
wire [nl2-1:0] ind1, ind2;
wire [nl2-2:0] tind;
wire [sts-1:0] pos;

//**********indice selection**************
wire [nl2-1:0] mask;
assign pos=stage-1,
 mask=({nl2{1'b1}}>>pos)<<pos,

 ind1 = ((cnt&mask)<<1)|(cnt&(~mask)),
 ind2 = ind1|(1'b1<<pos),

 tind=cnt<<(nl2-stage);

//**********butterfly input selection *************
assign bfin1 = data[ind1],
 bfin2 = data[ind2],
 tf=lut[tind];

//**********bit reversal**************
wire [nl2-1:0] c[n-1:0];
wire [nl2-1:0] r[n-1:0];
genvar k,l;
generate
 for(k=0;k<n;k=k+1)
  begin:loop1
   assign c[k]=k;
   for(l=0;l<nl2;l=l+1)
```

```
      begin:loop2
        assign r[k][l]=c[k][nl2-1-l];
      end
    end
endgenerate



//*********power value**************
wire [15:0] val;
wire signed [7:0] re, im;
wire signed [15:0] re2, im2, sum;
wire [7:0] aval;
wire sc;

assign sc=(stage==nl2+2);
assign val = data[cnt];
assign re[7:0] = sc? data[n/2][15:8]:val[15:8]<<1,
  im[7:0]=sc? data[n/2][7:0]: (val[7:0]<<1),
  re2=re,
  im2=im,
  sum=re2*re2+im2*im2,
  aval=sum[14]?sum[14:6]-1:sum[14:6];

//********output calculation*************
reg [15:0] summer;
reg [7:0] outr[7:0];
wire [15:0] summed;
assign summed = summer + aval;


//final output
reg [ofd+2:0] ocount;
assign clk = ocount[cfd];
assign sclk = ocount[sfd];
wire [7:0] cur;
wire [2:0] cc;
assign cc= ocount[ofd+2:ofd],
  cur = outr[cc],
  column = 8'b10_00_00_00>>cc;

assign row[7]=~cur[7];
genvar m;
generate
```

```
 for(m=0;m<7;m=m+1)
  begin:loop
   assign row[m]=row[m+1]&(~cur[m]);
  end
endgenerate



//**********initial state**************
initial
begin
 state=(nl2+1)<<(nl2-1);
 ocount=0;
 for (i=0;i<n;i=i+1)
  begin
   samples[i]=0;
                    data[i]=0;
  end
end



always @(posedge clkin)
 begin
  ocount <= ocount + 1;
 end


//**********next state logic**************
always @(posedge clk)
 begin


  case (stage)
   0:
    begin
     for(i=0;i<n;i=i+1)
      begin
       data[c[i]]<={samples[r[i]], 8'b0};
      end

     state[nl2+sts-2:nl2-1]<=1;
     state[nl2-2:0]<=0;
```

```
  end

4'd7:
  begin
   outr[7]<=summed;

   state<=0;
  end

4'd6:
  begin
   case (cnt)
     0:
      summer<=aval;
     2:
      begin
       outr[0]<=summed;
       summer<=0;
      end
     4:
      begin
       outr[1]<=summed;
       summer<=0;
      end
     6:
      begin
       outr[2]<=summed;
       summer<=0;
      end
     8:
      begin
       outr[3]<=summed;
       summer<=0;
      end
     10:
      begin
       outr[4]<=summed;
       summer<=0;
      end
     12:
     begin
       outr[5]<=summed;
       summer<=0;
      end
     14:
```

```
            begin
              outr[6]<=summed;
              summer<=0;
            end

          default:
            summer<=summed;

        endcase
        state <= state + 1;

      end

    default:
      begin

        for(i=0;i<n;i=i+1)
          if(ind1==i)
            data[i]<=bfout1;
        for(i=0;i<n;i=i+1)
          if(ind2==i)
            data[i]<=bfout2;

        state<=state+1;
      end

   endcase
  end

 //**********sampling block**************
 always @(posedge sclk)
  begin

   samples[0] <= in-8'd128;
   for (i=1;i<n;i=i+1)
     samples[i]<=samples[i-1];
  end


endmodule
```

```
module butterfly(in1, in2, tf, out1, out2);

 input [15:0] in1, in2, tf;
 output [15:0] out1, out2;

 wire signed [7:0] in1r, in1i, in2r, in2i, tfr, tfi,out1r, out1i, out2r, out2i;
 wire signed [15:0] i1r, i1i, i2r, i2i, tr, ti, t1r, t1i, t2r, t2i, o1r, o1i, o2r, o2i;

 assign in1r[7:0] = in1[15:8],
  in1i[7:0] = in1[7:0],
  in2r[7:0] = in2[15:8],
  in2i[7:0] = in2[7:0],
  tfr[7:0] = tf[15:8],
  tfi[7:0] = tf[7:0],
  out1[15:8] = out1r[7:0],
  out1[7:0] = out1i[7:0],
  out2[15:8] = out2r[7:0],
  out2[7:0] = out2i[7:0];

 assign i1r=in1r,
    i1i=in1i,
    i2r=in2r,
    i2i=in2i,
    tr=tfr,
    ti=tfi;

 assign t1r=i1r<<7,
  t1i=i1i<<7;

 assign t2r=i2r*tr-i2i*ti,
  t2i=i2r*ti+i2i*tr;

 assign o1r=t1r+t2r,
  o1i=t1i+t2i;

 assign o2r=t1r-t2r,
  o2i=t1i-t2i;

 assign out1r[7:0]=o1r[15:8],
  out1i[7:0]=o1i[15:8];

 assign out2r[7:0]=o2r[15:8],
  out2i[7:0]=o2i[15:8];

endmodule
```

Comments:

(i) We depended on the synthesizer to implement arithmetic circuits like adders and multipliers and also few other modules like multiplexers, shifters.

(ii) The Spartan 3E FPGA has built in DSP blocks which can effectively implement the adders and multipliers.

(iii) We used array indexing to implicitly implement multiplexers.

Limitations and Scopes of Improvement:

(i) Our HDL code is not optimized. As beginners we took on the daunting task of implementing a complicated algorithm using HDL. We could not implement a bus system and ended up using too many multiplexers. As a result we could only process 32 samples at a time using the resources of the FPGA. Thus we had a low resolution for the frequency domain.
This limitation can be overcome by further optimizing the code and implementing buses.

(ii) We used ADC 0804. This ADC is not suited for audio sampling as it has a sampling rate of only 10 kHz. So we could only include frequencies up to 5 kHz. Nevertheless we used this ADC because it has a simple interface.
We could improve our design by using an ADC with sampling rate at least 40 kHz which is suitable for audio sampling.

(iii) Although we intended to use an amplifier in the analog input stage, we excluded it from our final circuit for simplicity.
Adding an amplifier would result in a better design.

(iv) We used fixed point numbers because they are easy to operate on. Using floating point numbers would require arithmetic operations to be performed sequentially. But as beginners we cannot afford this complicated design.
A better design can be achieved by using floating point numbers.

(v) Our LUT contains N/2 data. But actually we could store only N/8 data. The other data could be reproduced from these values.

(vi) We used a counter to divide clock frequencies. A better design would be to use built in DSMs (Digital Signal Modulator) of the FPGA for this task.

(vii) The Spartan 3E FPGA is not suitable for complex DSP processors. A higher end FPGA would have more resources for implementation of a better design.

(viii) We used too many combinational blocks which used up the resources of the FPGA.
Performing some operations sequentially would result in a better optimized design.

If the stated limitations could be overcome we would be able to sample the input at a higher frequency process a higher number of samples thus covering the whole audible range of frequencies at a higher resolution. We would also be able to increase the number of bands in the spectrum analyzer.