

Audio Band Digital AM Demodulator

Final Project Report

December 10, 2006

E155

Eddy Chavarria and Danny La Valle

Abstract:

AM modulation is often used for radio signals. Most AM radio receivers use analog circuitry to demodulate the received signal. This project develops a proof of concept for demodulating signals digitally. The system digitizes an analog input signal with a frequency range of 0 to 20 kHz using a PIC microcontroller. Demodulation is accomplished through a digital signal mixer to down convert a received signal frequency and a digital low pass filter to remove unwanted image frequencies resulting from mixing. Both of these elements are implemented on a FPGA. A digital to analog converter is used to produce an analog output allowing demodulated audio signals to be played back through a speaker.

Introduction

Amplitude modulation is commonly used when broadcasting radio signals. Most radio receivers use analog circuitry to demodulate and recover the signals they receive. This project develops basic digital hardware as a proof of concept to demodulate AM signals digitally.

The system built is a digital AM demodulator capable of recovering and playing small bandwidth audio signals modulated at either 10 kHz or 14 kHz. The system is specifically designed to work with electronic music devices that output analog voltage signals in the range of -2 V to 2 V.

The analog input signal is digitized using the PIC microprocessor. The PIC's analog to digital converter (A/D) has a sampling rate of 50 kHz and uses a 10 bit resolution per sample. The samples are passed from the PIC to the FPGA where the digital signal is processed.

The FPGA implements a digital signal mixer to down convert the received signal frequency and a low pass digital filter to remove unwanted image frequencies resulting after mixing. Both the signal mixer and the filter require fixed point multiplication and make use of the multipliers built into the FPGA. After each sample is processed by the FPGA, it is passed to the digital to analog converter (D/A).

The D/A is the only hardware used that does not come from supplies available in the lab. It receives 8 bit samples from the FPGA to produce an analog voltage output of the recovered audio signal. The analog signal is then amplified and the audio is played back through a speaker.

New Hardware

Digital to Analog Converter

The only new hardware used that was not available in the lab was a digital to analog converter (D/A) from Analog Devices, AD558KN. This device interfaces an 8-bit data bus to an analog system. In this case, it is used to interface the output of the FPGA with the audio amplifier and speaker. Figure 1 shows descriptions for each pin of this device.

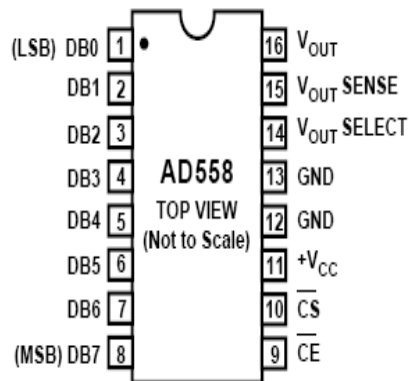


Figure 1 AD558KN Pin descriptions

This D/A requires no external components to function properly and it may operate with a supply voltage ranging from +5 V to +15 V. This device may also be configured to produce outputs in the ranges of 0 V to +2.5 V or 0 V to +10 V. Figure 2 shows how to set the output in either of these ranges.

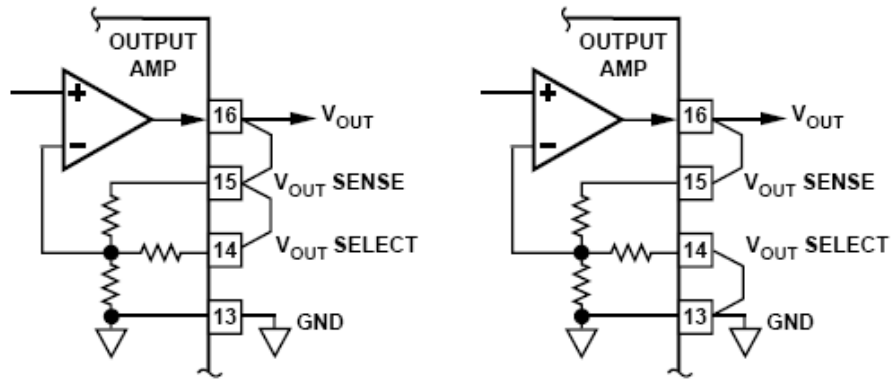
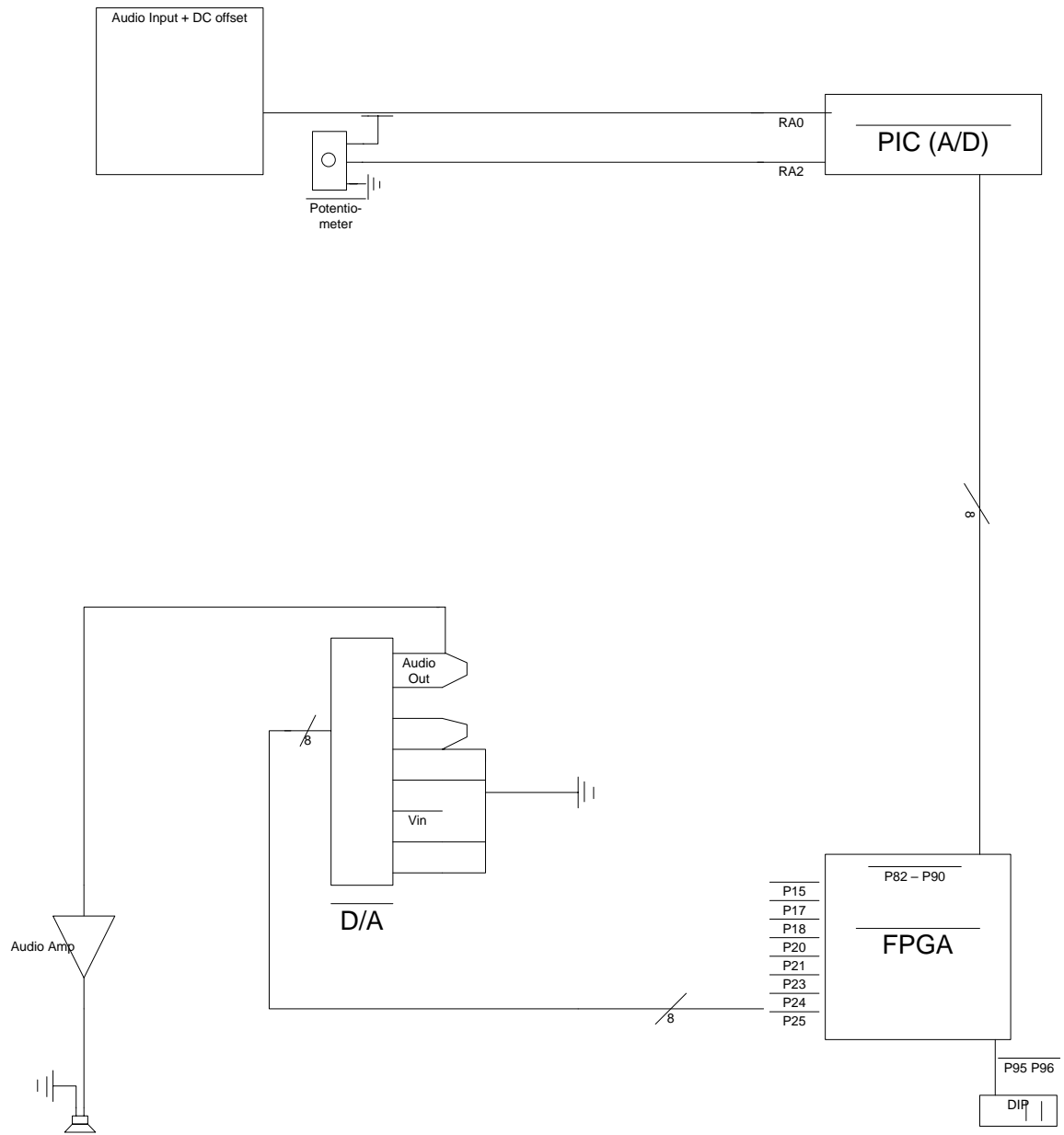


Figure 2
0 V to 2.5 V output (left) 0 V to 10 V output (right)

The digital input levels of the device are TTL/ low voltage CMOS. We found this device to work very well in the frequency ranges of 0 to 20 kHz (we didn't try anything higher than 20 kHz) and would recommend it for a similar application.

Schematics

Overall System Schematic



Microcontroller Design

The PIC microcontroller is used exclusively to digitize an analog input signal and pass the digital sample to the FPGA. The PIC is capable of sampling at 50 kHz with a 10 bit resolution.

The PIC routine waits until its analog to digital converter (A/D) module has acquired a sample. It then sends this sample to the FPGA over Port C. Because the digital to analog converter (D/A) uses 8-bits, only the top 8-bits from the A/D's 10-bit sample are used. After passing the sample to the FPGA, the PIC then restarts the A/D process and waits again until another sample has been acquired.

The PIC's A/D module does not accept negative voltage levels. Since the devices delivering the signal to the PIC do use negative voltages, it is necessary to first bias the incoming signal with a DC offset. For -2 V to 2 V source devices, an offset of 2 V is adequate; however, the default reference voltage for the PIC's A/D is set to VCC, or 3.3 V. If left alone, the PIC will clip part of its input. To prevent this from happening, the PIC is configured to use pin RA3 as reference voltage and a potentiometer is used to supply RA3 with 4 V.

FPGA DESIGN

We used the FPGA as a digital signal processor; it did all the digital mixing and filtering we needed to demodulate our signal. The FPGA takes as inputs the 8 bit data sample from the PIC, as well as a 2 bit select signal to choose which of the digital mixers the user wants to use. The clock input to the FPGA is not used to clock the modules; instead, we generate a 50 kHz clock by dividing the input clock and feed that signal to the rest of our design. The sample is converted to twos compliment using the *tcomp* modules which simply inverts the most significant bit. This was done so that we could multiply both positive and negative numbers on our FPGA. The *dflop* modules are simply just D Flip-Flops. The major modules of our design are the mixer modules and the filter module. Figure 3 shows the data path from the input to the mixers.

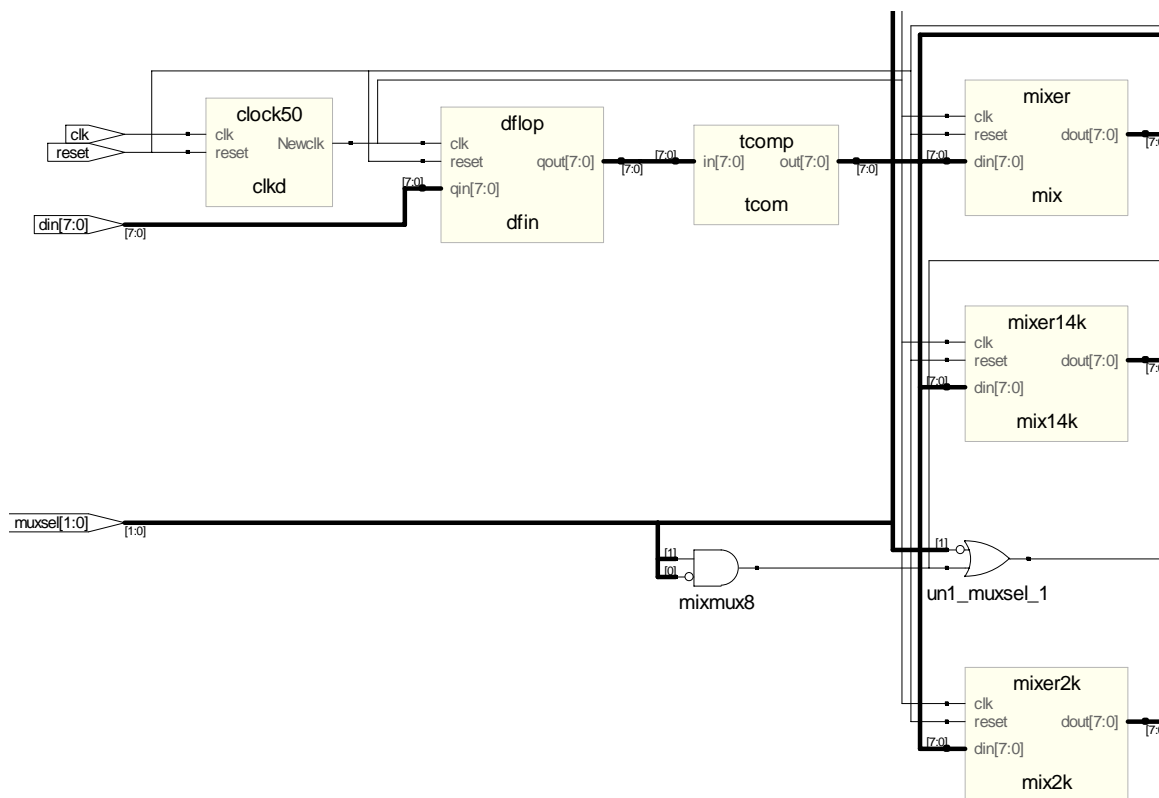


Figure 3 Data path of input to mixers

Mixer Module (mixer, mixer14k, mixer2k)

The mixer module is used to shift the frequency of the input signal. It does this by multiplying the incoming signal with a sinusoid at a given frequency. The result is a signal with frequency content shifted by the frequency of the sinusoid. The mixer uses stored values of a sinusoid sampled at 50 kHz to multiply with the input samples. Each of the different mixer modules, *mixer*, *mixer14k*, and *mixer2k*, use look-up tables of sinusoids of different frequencies but all sampled at 50 kHz. The three mixer modules allow the user to select sinusoids of 10 kHz, 14 kHz and 20 kHz respectively.

Filter Module (firlpnar)

In order to try and cut out unwanted frequencies from our mixed signals, we decided to implement a low-pass filter on our FPGA. We decided to implement a time domain convolution to filter our signal, instead of dealing with a complicated Fast Fourier Transform. We used a FIR filter, which we designed with Matlab. The filter coefficients were generated using the FDA filter design tool and correspond to the coefficients of a first-order difference equation. The Matlab FDA tool is straight forward to use, one just inputs the desired filter performance, such as stop and pass band, and the program generates the coefficients that describe such a filter. The coefficients generated are all decimal values, and thus necessitates that they be converted to signed 8-bit binary numbers. The difference equation that describes the filtered output is of the form

$$y[n] = \sum_{k=-N}^M b_k x[n-k]$$

The filter that we designed was a thirty-second ordered filtered, requiring 32 coefficients.

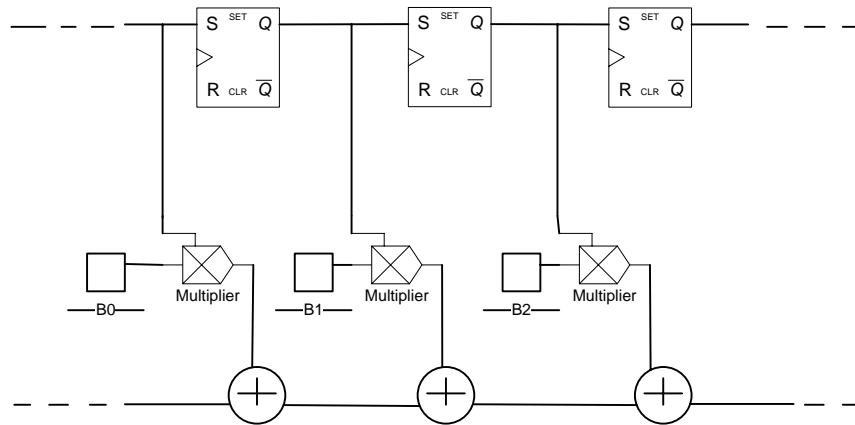


Figure 4 Filter Block Diagram

From Figure 4, we can see the way in which we implemented the filter. It is a series of Flip-Flops that hold the data samples as they flow into the filter, each data sample is multiplied by its corresponding coefficient and all added together. New data flows into the filter at every clock cycle, and an output is also generated every clock cycle.

The digitally filtered signal is then taken out of two's complement form, and output to the digital to analog converter. Figure 5 shows the data path from the filter to the output of the FPGA.

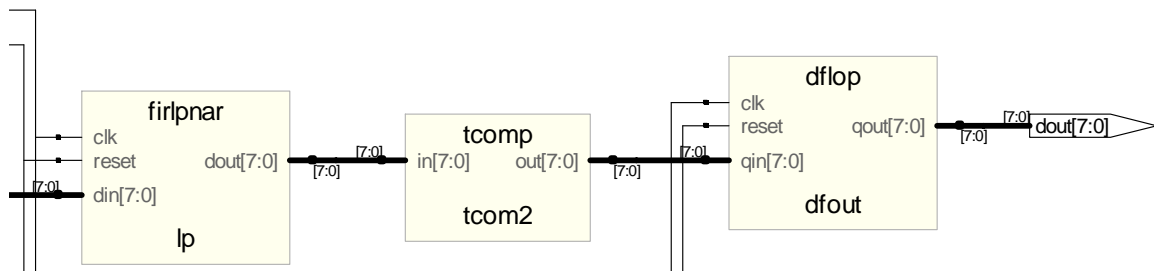
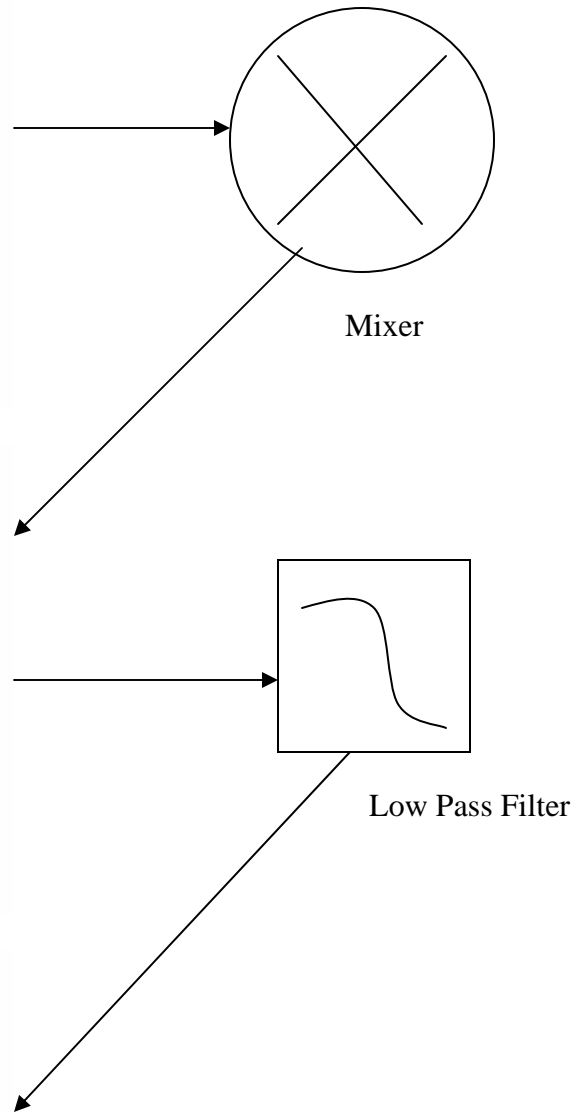
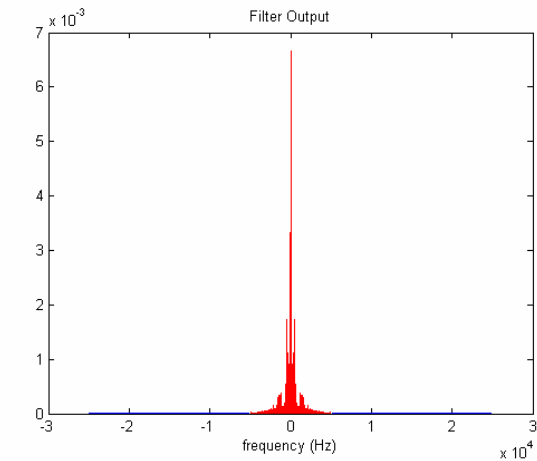
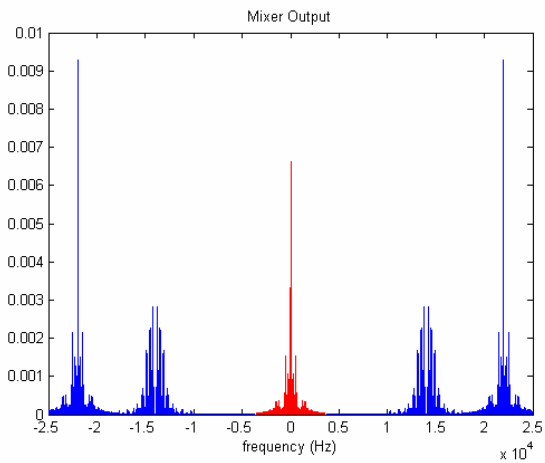
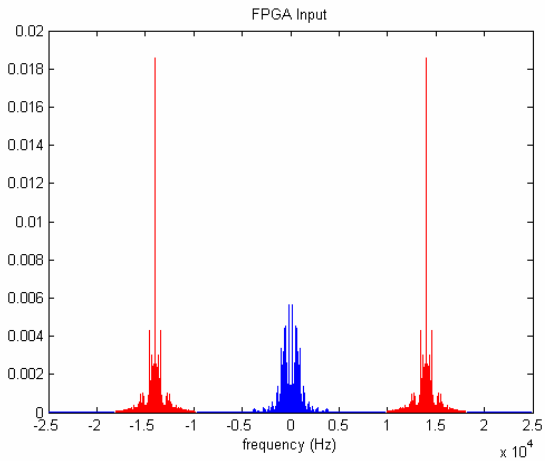


Figure 5 Filter output data path

FPGA Signal Flow

The following diagram shows the simulated results that the major components of the FPGA are expected to perform. The red spectrum indicates the signal of interest.



Results

Our design was able to filter and shift the frequencies of an input signal. However, the resulting audio suffered from much noise. We believe that both mixing and filtering are responsible for producing the observed noise.

We have found problems with our filter coefficients; namely, that some errors were made when converting the coefficients from decimal to a signed magnitude binary form. Passing a base band audio signal through the filter without any mixing produces an audio signal with some added noise, however this does not account for the distortion we observe when demodulating modulated signals.

The mixers have their own problems. They are particularly prone to producing an output that is clipped. Because the digital to analog converter can only use 8-bits, each multiplication must be truncated to 8-bits. This poses some problem when the multiplication is either too large or too small. If not enough significant bits are used in the truncated 8-bits then clipping occurs. If too many significant bits are used, signal resolution is lost.

We believe that the system can do better due to results we found using simulations in Matlab software. However, the results we did observe were encouraging that our system was performing the general operations we had designed it to perform.

Our original proposal consisted of an AM radio receiver and a digital demodulator. The goal of the project was to receive and digitize the entire AM band and digitally tune to a specific station. Two primary factors led us to revise the original design. The first was the complexity of designing the front-end radio receiver. We decided the time necessary to complete this part of the project would exceed the time

available in the semester. We chose to replace the front-end with an audio source that had an AM modulated signal we could generate ourselves. Secondly, the sampling rate of the PIC processor was greatly over estimated. The true sampling rate reduced the size of the band we were capable of sampling to roughly the audio band. This made changing the front-end to an audio source even more attractive. The changes made allowed us to implement a digital AM demodulator in a timely fashion using the hardware available.

References

[1] Using Embedded Multipliers in Spartan-3 FPGAs

<http://direct.xilinx.com/bvdocs/appnotes/xapp467.pdf>

[2] Wakerly, John F. *Digital Design Principles and Practices*. Upper Saddle River, NJ.

Prentice Hall. 2006

[3] Oppenheim, Alan. *Signals and Systems*. Upper Saddle River, NJ. Prentice Hall.

1997

Part List

Part	Source	Vendor Part #	Price
Audio Amplifier	Micro P's Lab		
D/A	Digi-Key	AD558KN	\$14.99
Potentiometer	Micro P's Lab		
Speaker	Micro P's Lab		

APPENDICES

Appendix A: Top Level Verilog Module (demod.v)

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Create Date:      23:16:27 11/27/2006
// Module Name:      demod
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module demod(clk, reset, din, dout, muxsel, sinout,clkout);
    input clk;
    input reset;
    input [7:0] din;
    output [7:0] dout;
    input [1:0] muxsel;
    output [7:0] sinout;
    output clkout;

    wire [7:0] dfinout;
    wire [7:0] lpout;
    wire [7:0] mixout;
    wire [7:0] tcout;
    wire      Newclk;
    wire [7:0] temp;
    wire [7:0] temp2;
    wire [7:0] magout;
    wire [7:0] mixout2k;
    wire [7:0] mixout14k;
    reg [7:0] mixmux;

    always @ ( * ) begin
        if ( muxsel == 2'b00 )
            mixmux = tcout;
        else if ( muxsel == 2'b01 )
            mixmux = mixout;
        else if ( muxsel == 2'b10 )
            mixmux = mixout14k;
        else
            mixmux = mixout2k;
    end

    end

    clock50 clkd(clk,reset,Newclk);
    dflop dfin(Newclk, reset, din, dfinout);
    tcomp tcom(dfinout, tcout);

    mixer mix(Newclk, reset, tcout, mixout);
    mixer2k mix2k(Newclk, reset, tcout, mixout2k);
    mixer14k mix14k(Newclk, reset, tcout, mixout14k);

    fir1pnar lp(Newclk, reset, mixmux, lpout);

    tcomp tcom2(lpout, magout);
    dflop dfout(Newclk, reset, magout, dout);

endmodule
```



```

// Create Date:      22:04:41 11/29/2006
// Module Name:      tcomp
////////////////////////////////////////////////////////////////
module tcomp(in, out);
    input [7:0] in;
    output [7:0] out;

    assign out = {~in[7], in[6:0]};

endmodule

```

Appendix E: 10 kHz Mixer Verilog Module (mixer.v, mixer14k.v, mixer2k.v)

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////
// Create Date:      23:42:12 11/27/2006
// Module Name:      mixer
////////////////////////////////////////////////////////////////
module mixer(clk, reset, din, dout);
    input clk;
    input reset;
    input [7:0] din;
    output [7:0] dout;

    wire [7:0] sout;

    sinelut lut(clk, reset, sout);
    spmult mul(din, sout, dout);

endmodule

```

Appendix H: Multiplier Verilog Module (spmult.v)

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////
// Create Date:      21:46:20 11/29/2006
// Module Name:      spmult
////////////////////////////////////////////////////////////////
module spmult(sample, c, out);
    input [7:0] sample;
    input [7:0] c;
    output [7:0] out;

    wire [7:0] magsam;
    wire [7:0] magc;
    wire [16:0] magout;

    assign magsam = sample[7] ? (~sample) + 1 : sample;
    assign magc = {1'b0, c[6:0]};

    assign magout = magsam*magc;

    assign out = sample[7]^c[7] ? ( (~magout[12:7]) + 1 ) :
magout[12:7];

```



```
endmodule
```

Appendix I: Sine Look-Up Verilog Module (sinelut.v, sinelut14k.v, sinelut.20k)

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////
// Create Date:      11:57:55 11/28/2006
// Module Name:      sinelut
////////////////////////////////////////////////////////////////
module sinelut(clk, reset, dout);
    input clk;
    input reset;
    output [7:0] dout;

    parameter s0 = 8'b0000_0000;
    parameter s1 = 8'b0111_1010;
    parameter s2 = 8'b0100_1011;
    parameter s3 = 8'b1100_1011;
    parameter s4 = 8'b1111_1010;

    reg [7:0] w0, w1, w2, w3, w4;

    assign dout = w0;

    always @ (posedge clk, posedge reset)
        if(reset) begin
            w0 <= s0;
            w1 <= s1;
            w2 <= s2;
            w3 <= s3;
            w4 <= s4;
        end
        else begin
            w0 <= w4;
            w1 <= w0;
            w2 <= w1;
            w3 <= w2;
            w4 <= w3;
        end
end

endmodule
```

Appendix I: Low Pass Filter Verilog Module (firlpnar.v)

```
`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
// Create Date:      23:27:40 11/27/2006
// Module Name:      firlp
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
module firlpnar(clk, reset, din, dout);
    input clk, reset;
    input [7:0] din;
    output [7:0] dout;
endmodule
```

```

parameter b0 = 0;
parameter b1 = 0;
parameter b2 = 0;
parameter b3 = 0;
parameter b4 = 0;
parameter b5 = 0;
parameter b6 = 8'b10000001;
parameter b7 = 8'b10000011;
parameter b8 = 8'b00000100;
parameter b9 = 8'b00000100;
parameter b10 = 8'b00000010;
parameter b11 = 8'b00000001;
parameter b12 = 8'b00001000;
parameter b13 = 8'b00001111;
parameter b14 = 8'b00010110;
parameter b15 = 8'b00011001;
parameter b16 = 8'b00011001;
parameter b17 = 8'b00010110;
parameter b18 = 8'b00001111;
parameter b19 = 8'b00001000;
parameter b20 = 8'b00000001;
parameter b21 = 8'b10000010;
parameter b22 = 8'b10000100;
parameter b23 = 8'b10000100;
parameter b24 = 8'b10000011;
parameter b25 = 8'b10000001;
parameter b26 = 0;
parameter b27 = 0;
parameter b28 = 0;
parameter b29 = 0;
parameter b30 = 0;
parameter b31 = 0;

wire [7:0] mul1, mul2, mul3, mul4, mul5, mul6, mul7, mul8, mul9,
mul10,
mul11, mul12, mul13, mul14, mul15, mul16, mul17,
mul18, mul19, mul20;

reg [7:0] x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x10,
x11, x12, x13, x14, x15, x16, x17, x18, x19, x20,
x21, x22, x23, x24, x25, x26, x27, x28, x29;

spmult spmul1(x6, b6, mul1);
spmult spmul2(x7, b7, mul2);
spmult spmul3(x8, b8, mul3);
spmult spmul4(x9, b9, mul4);
spmult spmul5(x10, b10, mul5);
spmult spmul6(x11, b11, mul6);
spmult spmul7(x12, b12, mul7);
spmult spmul8(x13, b13, mul8);
spmult spmul9(x14, b14, mul9);
spmult spmul10(x15, b15, mul10);
spmult spmul11(x16, b16, mul11);
spmult spmul12(x17, b17, mul12);

```

```

spmult spmul13(x18, b18, mul13);
spmult spmul14(x19, b19, mul14);
spmult spmul15(x20, b20, mul15);
spmult spmul16(x21, b21, mul16);
spmult spmul17(x22, b22, mul17);
spmult spmul18(x23, b23, mul18);
spmult spmul19(x24, b24, mul19);
spmult spmul20(x25, b25, mul20);

assign dout = mul1 + mul2 + mul3 + mul4 +
              mul5 + mul6 + mul7 + mul8 +
              mul9 + mul10 + mul11 + mul12 +
              mul13 + mul14 + mul15 + mul16 +
              mul17 + mul18 + mul19 + mul20;

always @ (posedge clk, posedge reset)
  if (reset) begin
    x29 <= 0;
    x28 <= 0;
    x27 <= 0;
    x26 <= 0;
    x25 <= 0;
    x24 <= 0;
    x23 <= 0;
    x22 <= 0;
    x21 <= 0;
    x20 <= 0;
    x19 <= 0;
    x18 <= 0;
    x17 <= 0;
    x16 <= 0;
    x15 <= 0;
    x14 <= 0;
    x13 <= 0;
    x12 <= 0;
    x11 <= 0;
    x10 <= 0;
    x9  <= 0;
    x8  <= 0;
    x7  <= 0;
    x6  <= 0;
    x5  <= 0;
    x4  <= 0;
    x3  <= 0;
    x2  <= 0;
    x1  <= 0;
    x0  <= 0;

  end
  else begin
    x29 <= x28;
    x28 <= x27;
    x27 <= x26;
    x26 <= x25;
    x25 <= x24;
    x24 <= x23;
    x23 <= x22;
  end

```

```
x22 <= x21;
x21 <= x20;
x20 <= x19;
x19 <= x18;
x18 <= x17;
x17 <= x16;
x16 <= x15;
x15 <= x14;
x14 <= x13;
x13 <= x12;
x12 <= x11;
x11 <= x10;
x10 <= x9;
x9 <= x8;
x8 <= x7;
x7 <= x6;
x6 <= x5;
x5 <= x4;
x4 <= x3;
x3 <= x2;
x2 <= x1;
x1 <= x0;
x0 <= din;
```

```
end
```

```
endmodule
```

Appendix J: PIC C Code

```
#include <p18f452.h>
#include <timers.h>

void main(void);
void isr(void);

short DUTYL;
short DUTYH;
short mynum;

#pragma code low_vector = 0x18
void low_interrupt(void) {
    _asm
        GOTO isr
    _endasm
}

#pragma code

void main(void) {
    // setup A/D
    ADCON0 = 0x85;
    ADCON1 = 0x21;

    TRISA = 0xFF;
    TRISC = 0;
```

```
PORTC = 0;

while(1) {
    if( ADCON0 == 0x81 ){
        ADCON0 = 0x85;
        PORTC = ADRESH;
    }
}
```