

# Polyphonic MIDI Synthesizer

Final Project Report

December 10, 2009

E155

Jackie Lam and Ian Bullock

## **Abstract:**

Earlier microprocessor labs demonstrated the simplicity of single note, square wave musical synthesis. However, real synthesizers are capable of playing multiple notes at once, and can use a variety of waveforms or samples. Our project is to create a single-sample based polyphonic MIDI synthesizer with envelope capabilities. A PIC microcontroller was used for initial MIDI signal processing and an FPGA was used to store the note waveform and produce 8-bit audio output with up to eight notes of polyphony. An envelope is applied to each note so that the loudness of the note varies based on how recently it was struck or released. Although the output sound of the final device deviates slightly from the expected sound, a functional, pleasant sounding, 8-note polyphonic MIDI synthesizer was successfully produced.

# Table of Contents

I.	Introduction / Problem Statement .....	3
A.	Motivation .....	3
B.	Overall System .....	3
II.	New Hardware.....	3
A.	DAC .....	3
B.	Low Pass Filter .....	4
III.	Hardware Layout.....	4
IV.	Microprocessor Design .....	6
V.	FPGA Design.....	7
A.	Eight Port Memory .....	9
B.	Envelope Generation.....	10
VI.	Results .....	10
VII.	References.....	11
VIII.	Parts List .....	12
IX.	Appendix: .....	13
A.	Schematic .....	13
B.	Verilog .....	14
C.	PIC Code .....	14

# I. Introduction / Problem Statement

## A. Motivation

Electronic synthesizers have been used for several decades to produce a wide range of musical sounds. Although originally complex analog systems, it is now possible to implement a synthesizer on a single chip. This project is an exploration in efficient implementation of a functional digital polyphonic synthesizer under the limitations of the FPGA used.

## B. Overall System

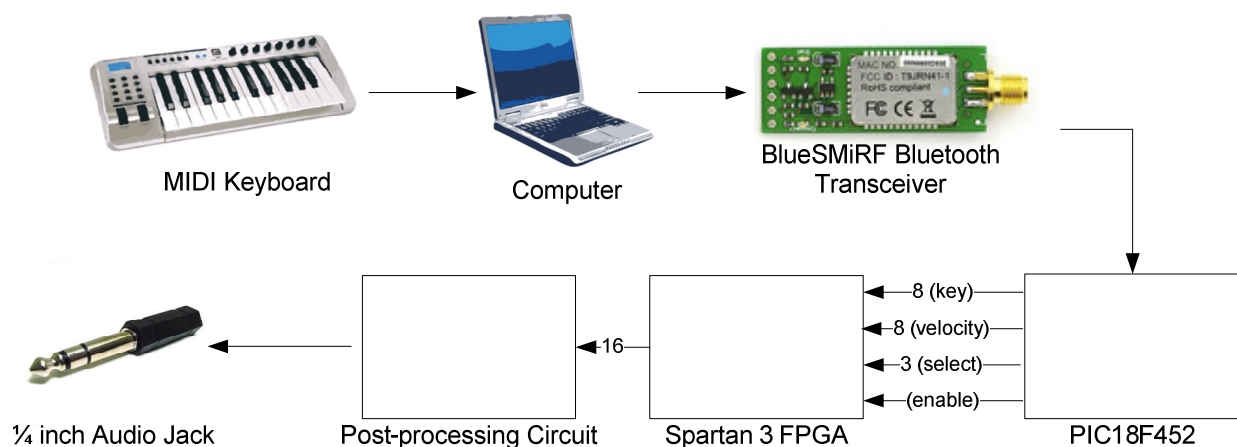


Figure 1: Block diagram of overall system

The overall goal of our project is to produce a polyphonic MIDI synthesizer. Using a USB MIDI input device such as a keyboard, a computer sends the MIDI data serially over a Bluetooth link. A PIC18F4520 microcontroller captures the note on/off messages sent and controls a Spartan 3 FPGA which acts as a synthesizer. The FPGA has eight note generators for eight simultaneous notes, and produce a 16-bit digital audio output at 71.125 kHz. In our project, the most significant bits of the audio signal are connected to an 8-bit DAC and the audio is filtered to produce the final analog output.

# II. New Hardware

## A. DAC

A component of the project consists of an 8-bit digital-to-analog converter (DAC). The DAC we've selected is the AD 558JN. The 558JN can be configured to have an output range of 0V to 2.56V or 0V to 10V. For our purpose, the 0V to 2.56V setting is selected by connecting  $V_{out}$  SELECT and  $V_{out}$

SENSE to  $V_{out}$ . The FPGA outputs an 8-bit digital audio signal to the DAC at pins 1-8 and the chip outputs the analog audio signal on pin 16.

## B. Low Pass Filter

To filter out the high frequency noise from the analog signal, two active low-pass filters are connected in series. The low-pass filters consist of a  $0.033\mu\text{F}$  capacitor,  $1000\Omega$  resistor, and a LM741 operational amplifier. The schematic of the circuit is as follows:

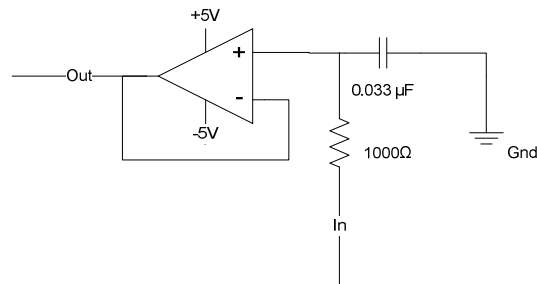
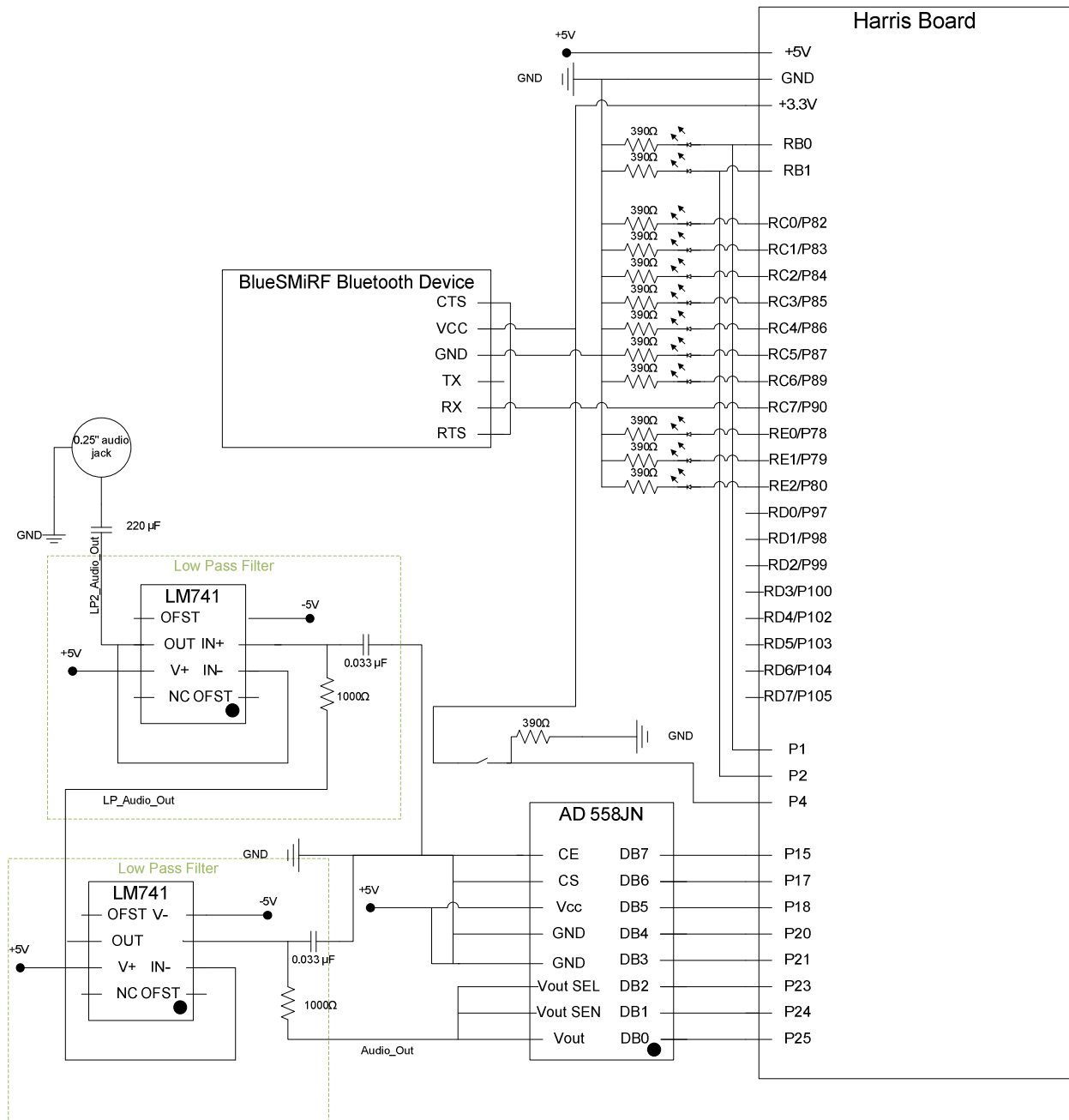


Figure 2: Low-pass filter

With the selected resistor and capacitor, the frequency response of the circuit has a corner frequency of  $4825\text{Hz}$ , a little above the top piano key frequency. Beyond the corner frequency, the response drops off  $20\text{dB}$  per decade corresponding to  $6\text{dB}$  per octave.

## III. Hardware Layout

Before the MIDI messages are inputted to the PIC microcontroller, the keyboard MIDI signals are relayed through a computer utilizing a third party software, SpikenzieLab Serial – MIDI Converter: v2c. Through a USB connection, the keyboard transmits the key presses to the computer. The software directly relays the data to the outgoing COM port of the attached Bluetooth dongle.



**Figure 3: Breadboard schematic**

The BlueSMiRF Bluetooth transceiver transmits the MIDI data to the PIC microcontroller using the USART protocol set up for 115.2k baud. The BlueSMiRF device is configured to perform its own handshaking; CTS and RTS are directly connected. The PIC program then extracts the key and velocity information of an incoming note and also assigns the note to one of the eight note generators. The 7-bits key information, 7-bits velocity information and 3-bits select are displayed on LEDs along with the 1-bit on/off and 1-bit enable signal. The data is also sent to the FPGA. The

FPGA processes the information and outputs an 8-bit digital audio signal at 78.125 kHz. Note that an external reset switch is tied to pin 4 of the FPGA for the program reset.

From the output of the FPGA, the digital audio is converted to analog through the 558JN as mentioned in Section II.A. The outputted analog signal is filtered through a series of two RC low pass filters. The filters have a corner frequency of 4825Hz. The two filters attenuate the high frequency noise from the audio signal by dropping 12dB per octave above the corner frequency. The filtered audio signal is passed through a 220µF capacitor to remove the DC component and then is connected to a ¼ inch audio jack.

## IV. Microprocessor Design

The main function of the PIC program is to decode the input MIDI signals and output note-on and note-off information. Additionally, the program acts as a controller and assigns incoming notes to the eight different note generators in the FPGA. The system utilizes the BlueSMiRF Bluetooth device to input the MIDI signal in serial through the USART port, RC7 of the PIC microcontroller. The program continuously checks the incoming bytes for the byte corresponding to note-on and note-off commands. Once detected, the program extracts the next two bytes for key and velocity information. A function is called to either assign the note to a note generator if the message is a note-on or remove the assignment if it is a note-off. Lastly, the enable bit is pulsed to notify the FPGA to update the assigned note generators.

MIDI messages are variable in length. The messages are comprised of a command byte usually followed by one or two parameters for that command. It is important to note that all command bytes begin with a '1' in the most significant bit and all parameters range from 0 to 127, therefore having a '0' in the most significant bit. For our project, the two MIDI commands of interest are note on and note off.

Command	Meaning	Parameter 1	Parameter 2
0x80	Note-On	Key	Velocity
0x90	Note-Off	Key	Velocity

Figure 4: MIDI note-on and note-off commands

At the start of the program, setup() is called. The function setup() configures the USART module on the PIC. The Transmit Status and Control Register (TXSTA) and the Receive Status and Control Register RCSTA) are set to 0x04 and 0x90, respectively. The settings enable the serial receiver using the high baud rate while keeping the transmitter disabled. The SPBRG register is set to 10 to match the 115200 baud rate of the BlueSMiRF. The setup() function also sets the corresponding pins on port B, C, D, and E as outputs, and initializes various variables.

Once initialized, the main program continuously receives MIDI data from the RC7 port. The program continues to wait in standby until a note-on or note-off command is received. When

detected, RB1 is set to 1 if note-on and set to 0 if note-off. Then the following two bytes, key and velocity, are written to port D and port B, respectively.

After the MIDI message information is extracted, `assign()` is called. `assign()` controls the note assignments to the eight note generators. If the command is a note-on, `assign()` appoints the key press to an unused note generator. If more than eight keys are pressed and all eight note generators are used, then `assign()` replaces the key pressed in the last note generator.

To do so, an array of eight elements corresponding to the eight note generators is used. The array elements are set to zero if the note generator is not being used. If it is in use, the corresponding element holds the key of the pressed note. During a note-on, the index is incremented until an empty element is found or until the end of the array. The resulting 3-bit index corresponds to one of the note generators and is assigned to port E, the select bits. During the note-off, the index is incremented again. The key from the command is compared to the elements of the array. When a match is found, the index is written to the select bits and the element is cleared, marking the note generator unused.

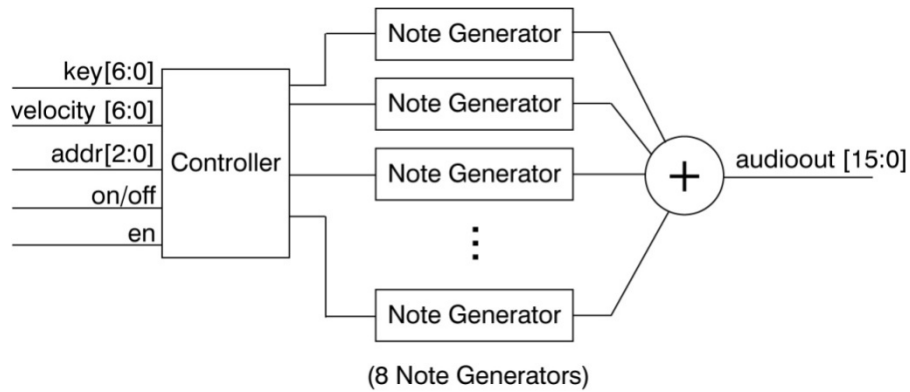
Lastly, an enable bit, RB0, is toggled. The rising edge of the enable signals the FPGA to sample the data and the MIDI message information is relayed to the FPGA. All the pins relaying information from the PIC to the Spartan3 FPGA is provided below.

Port	Pins	Value	Bitsize
B	0	Enable	1
B	1	On/Off	1
C	6-0	Velocity	7
D	6-0	Key	7
E	2-0	Select	3

Figure 5: PIC outputs to FPGA

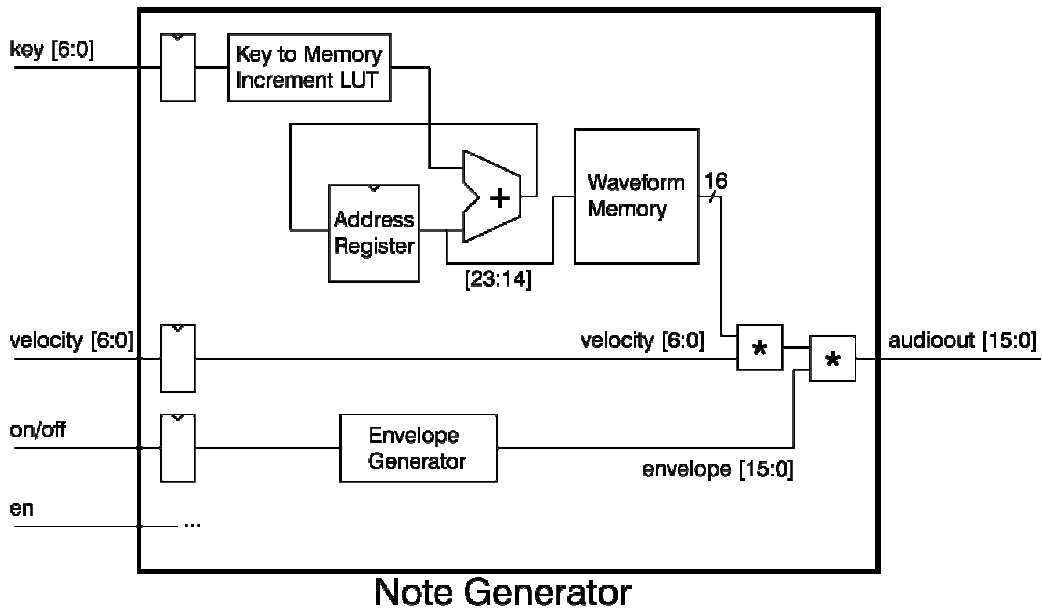
## V. FPGA Design

The overall function of the FPGA is to take in key, velocity, and on/off data from the PIC, and provide appropriate sound output with up to eight simultaneous notes. This is accomplished through using eight separate copies of a note generator module. The eight outputs from the note generators are summed and divided by eight to give the final audio output. Some controller logic is used to take the outputs of the PIC and convert them to appropriate enable signals for the note generators.



**Figure 6: Overall FPGA Design, showing eight note generators and some combinational logic that controls which generator is enabled.**

Each note generator contains several components. All note generators are clocked at the audio output frequency of 78.125kHz. The output frequency is determined by dividing the initial 20MHz clock by 256. The key and velocity inputs to each note generator are passed through registers enabled by the overall enable signal going to the generator. The key on signal controls the note envelope generator that has been recently added.



**Figure 7: Note Generator that generates the audio for a single note.**

The key data addresses a “key to memory increment” conversion look up table. This lookup table is used to determine how much to add to the current waveform memory address on each clock cycle. For example, a low pitched key will give a small memory increment value, making the note generator step through the stored waveform slowly. A high pitched key will result in faster stepping through the waveform.

Equation 1 below shows the method used to calculate the memory increment.



$$\text{Memory Increment} = \left( \frac{F_{\text{note}}}{F_{\text{clk}}} \right) \left( \frac{N \text{ samples}}{1 \text{ period}} \right) 2^{\text{number of decimal bits}} \quad [1]$$

$F_{\text{note}}$  is the frequency of the note being played,  $F_{\text{clk}} = 78125\text{Hz}$  is the frequency of the note generator clock, and  $N$  is the number of samples in one period of the sampled waveform. Since low pitched notes may only change samples every few clock cycles, 10 decimal bits were used to ensure accurate note pitches. A Java program was used to generate the Verilog code for the key to memory increment lookup table resulting from this equation. A Java program was also used to generate the Verilog code for a waveform ROM with a 128 sample per period sine wave.

The output of the waveform memory differs from the overall note generator output only in scaling. The memory output is first multiplied by the note envelope, and then by the note velocity to give the final output. Some care is required here to preserve 16'h7FFF as the center of the final output waveform. Two's complement arithmetic is used to initially shift down the waveform by 16'h7FFF. The waveform is then multiplied by the envelope and velocity, with appropriate right shifts on the results to make sure the final result will be a 16 bit number. The scaled waveform is then shifted back up by 16'h7FFF to restore its original center position. A little amplitude is lost in the normalization process, since for example multiplying by an all high 7-bits velocity is multiplying by  $2^7-1$ , while the following seven bit right shift will divide by  $2^7$ .

A final register was also added to the output of the note generator. Since a fair amount of combinational logic is used in the calculation of the final note output, the register helps ensure that varying delays in the combinational logic will not produce unintended momentary changes in the audio output.

### **A. Eight Port Memory**

One of the challenges in implementing eight note generators was to create an 8-port memory. This was done using a single port ROM and some additional logic running at the faster (20 MHz) clock rate. A continuously counting port register controls which of the ports is active during a given cycle of the 20 MHz clock. An eight input multiplexer controls which address is currently the ROM address input, and eight registers are used to store the output data for the eight ports.

The outputs of the 8-port ROM are temporarily incorrect for as much as eight clock cycles after a change in addresses on the slow clock edge, but the data will not be read until 256 clock cycles later due to the fast clock being divided by 256. Since the 8-port ROM interacts with two different clock domains, synchronizers were added to the slow-clock inputs to the memory. This is to ensure that new address data will not propagate to the memory outputs before the old memory data is read, especially in the case of skew between the slow and fast clocks.

## B. Envelope Generation

An envelope generator was created to vary the amplitude of a pressed note over time. The envelope can be divided into attack, delay, sustain, and release phases. In the attack phase, the note has just been pressed, and the note volume rises up to full amplitude. In the delay phase, the volume dies off slightly from the initial attack peak, eventually reaching the sustain volume. The sustain volume level is maintained until the note is released. When the note is released, its sound dies away based on the slope of the release section. The FPGA implementation of the envelope can be seen below in Figure 8.

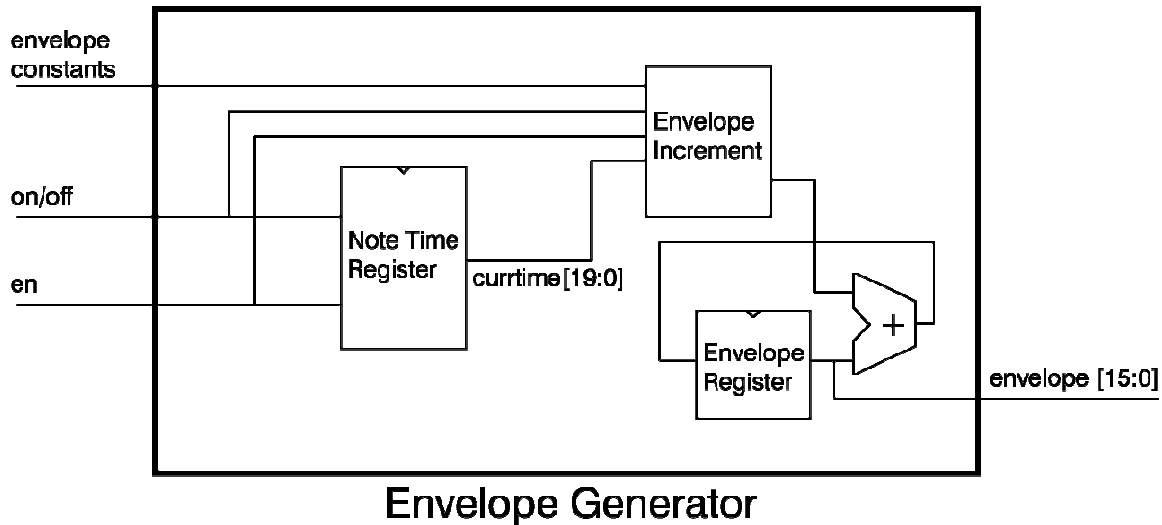


Figure 8: Envelope Generator used in each note generator to calculate the current envelope.

Several constants are input into the envelope generator. These constants, such as the attack slope or delay time, control the slopes and spacing of the envelope. The note time register keeps track of the time elapsed since the beginning of the note. This information, along with the constants and whether the note is on or off, determines what change will be made to the envelope register in the following clock cycle.

The envelope uses 2's complement "saturation" arithmetic. The 2's complement numbers allow the envelope to decrease during the decay and release sections of the envelope. The "saturation" adder used provides intelligent overflow – if the envelope is subtracted past 0, the output will just stay at 0, and if the output is added to past 16'hFFFF, the output will simply stay at 16'hFFFF. This modified adder helps to ensure that the envelope will behave as expected, especially in the condition that improper constants are used.

## VI. Results

Overall, the project was successful in satisfying the specifications in the project proposal. We produced a polyphonic synthesizer that can synthesize up to eight notes at 78.1kHz, based on inputs from a MIDI instrument. The MIDI instrument is connected via USB to a computer, which passes MIDI messages to the PIC via a BlueSMiRF Bluetooth® serial link. The PIC successfully picks

up MIDI messages and passes the appropriate control signals to the FPGA synthesizer module. Even after pressing quite a few keys in rapid succession, notes do not get stuck on or otherwise behave strangely. The envelope function was also implemented for each of the note generators, making the output amplitude of the note vary after the key is pressed, and avoiding abrupt note cutoffs.

Some challenges were encountered over the course of the project. Difficulties in implementing an optocoupler circuit led to an early revision of the project proposal to use the computer link instead. This decision was made to allow more focus on the other parts of the project.

The resource constraints of the FPGA presented challenges throughout the design. For example, some tasks that could be accomplished with a few multiplication and addition operations, such as key to memory increment conversion, were instead done with pre-computed lookup tables. Storing the waveform memory eight times would have been impractical, so an 8-port ROM module using a faster clock was created.

For some time, a mysterious high pitched hum was heard in the audio output of the system. It was eventually discovered that the hum was caused by the BlueSMiRF module on the breadboard. Maintaining a short, low impedance analog signal path, as well as grounding the analog output jack far away from the BlueSMiRF device greatly reduced the noise.

The output sound of the device is still not exactly what might have been expected. With the enveloping applied, the synthesizer sounds something like a slightly distorted pan-flute. Further work would be required to figure out the causes for all of these distortions, many of which seem to depend on the position within the envelope, or the pitch of the note. It is possible that there are some modifications necessary in the note scaling code, which may be providing erroneous output in a few rare cases. It is also possible that the audio output's distortions are due to the signal drifting outside the input range of the guitar amplifier being used. Despite these remaining puzzles, the output of the synthesizer is quite pleasant and useable as it is.

## VII. References

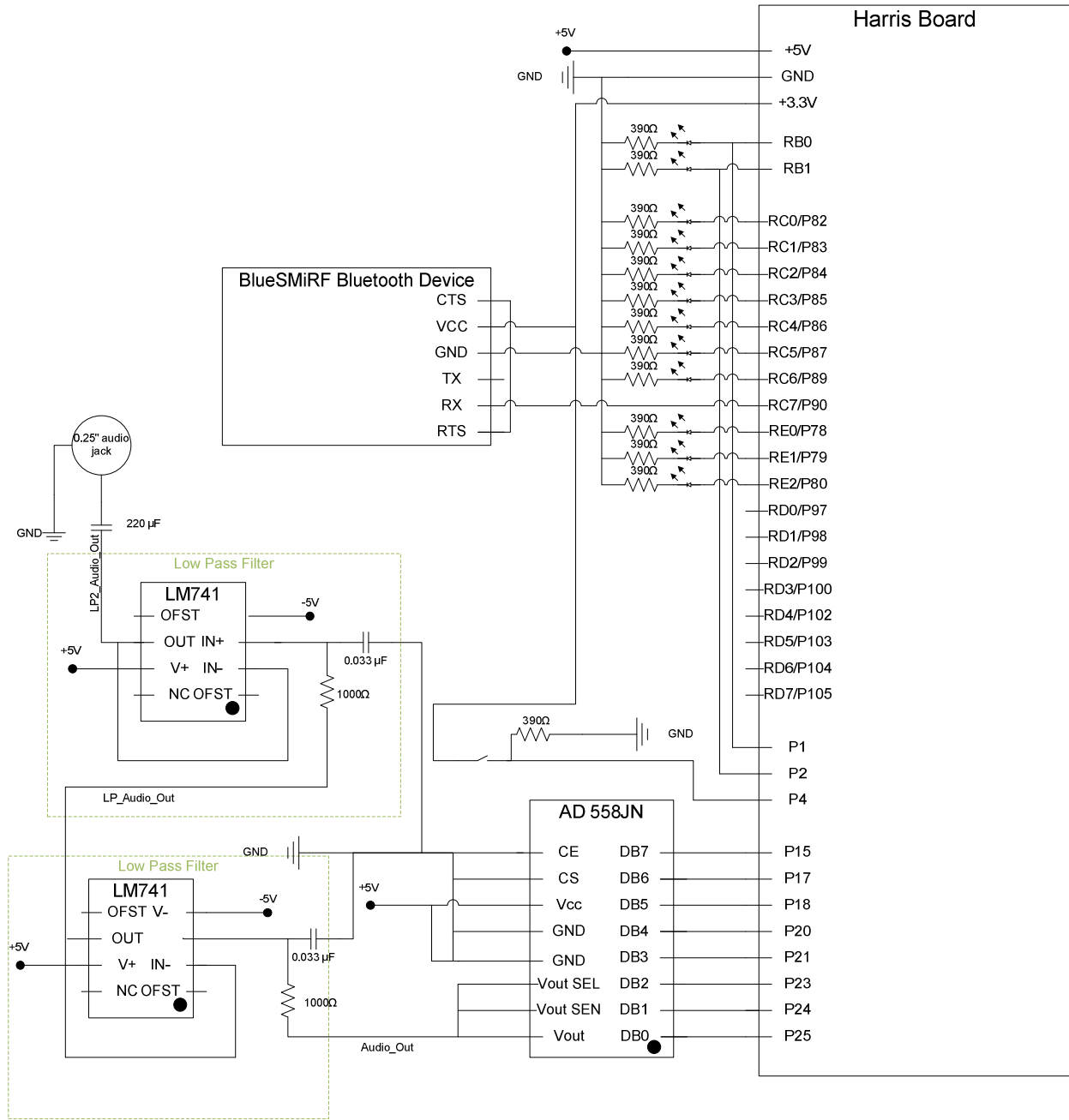
- [1] Craig Stuart Sapp, Essentials of the MIDI protocol, <<https://ccrma.stanford.edu/~craig/articles/linuxmidi/misc/essenmidi.html>>
- [2] Tutorial: MIDI and Music Synthesis, <[http://www.midi.org/aboutmidi/tut\\_midimusicynth.php](http://www.midi.org/aboutmidi/tut_midimusicynth.php)>

## VIII. Parts List

Quantity	Item	Price
1	8-bit digital-to-analog converter, AD 558JN	\$13.21
1	MIDI input jack	\$0.72
1	¼" phone jack	\$2.60
1	BlueSMiRF Bluetooth Modem	-
2	operational amplifier, LM741	-
12	LEDs	-
13	390Ω resistor	-
2	1000Ω resistor	-
2	0.033μF capacitor	-
1	220μF capacitor	-
1	push button	-

# IX. Appendix:

## A. Schematic



## B. PIC Code

```
// Final_jl.c
// Jackie Lam < flam@hmc.edu >
// Function: Takes a midi input and extract all the note on/off commands.
// Each on/off command will be outputted with its corresponding key,
// velocity, on/off, and an enable bit. Additionally, it will assign
// the note into an open "frequency generator". If all 8 "frequency
// generator is filled, the first note played will be replaced.

#include <p18f452.h>
#include <stdio.h>
#include <stdlib.h>

/* Function Prototypes */
void main(void); // main function
void setup(void); // initialize USART and I/O
void assign(char key); // assign notes to 1 of 8 FG
void reset(void); // reset PIC

/* Initialize Variables */
char address[8]; // maintain assignment of pressed keys

/* Main program */
void main(void){
    char i;
    char data;
    char key;
    char vel;
    setup();
    i=0;
    while (1){
        while(PIR1bits.RCIF==0){}
        data = RCREG;
        if ((data & 0xF0) == 0x80 || (data & 0xF0) == 0x90){
            PORTB=data>>3; // 'on/off' to PORTB.RB1
            while(PIR1bits.RCIF==0){}
            key = RCREG;
            PORTD=key; // 'key' to PORTD
            while(PIR1bits.RCIF==0){}
            vel = RCREG;
            PORTC=vel; // 'velocity' to PORTC
            assign(key);
            PORTBbits.RB0=1;
        }
    }
}

//initialize USART, I/O, and variables
void setup(void){
    char i;
    /*
    TXSTA settings, transmit status & control register
    7: 0, does not care
    6: 0, 8-bit transmission
    5: 0, transmit enable bit
    4: 0, asynchronous mode
    3: 0, unimplemented
    2: 1, high baud rate select
    1: 0, transmit shift register full
    0: 0, 9th bit of transmit data
    RCSTA settings, receive status & control register
    7: 1, enable serial ports
    6: 0, 8-bit reception
    5: 0, does not care
    4: 1, receive enable bit
    3: 0, address enable detection
    2: 0, framing error bit
    1: 0, overrun error bit
    */
}
```

```

    0: 0, 9th bit of receive data
*/
//SPBRG= 9; // SPBRG for MIDI (baud = 31250)
SPBRG =10; // SPBRG for Bluetooth
TXSTA= 0x04;
RCSTA= 0x90;
TRISB = 0xfc; // Configure B[0] & b[1] as output
TRISC = 0x80; // Configure input USART port & outputs
TRISD = 0x00; // Configure port D as output
TRISE = 0; // Configure port E pins as output
for(i=0; i<8; i++){ // Clear all addresses
    address[i]=0x00;
}
PORTE = 0;
}

// Address the key to 1 of 8 FG. It will assign it to an unused FG or replaces
// the last note.
void assign(char key){
    char i;
    i=0;
    if (PORTBbits.RB1==1){ // on detected: assign to a FG
        while((address[i]!=0x00) && (i<7)){i++;}
        address[i]=key;
        PORTE = i;
    }
    else{ // off detected: remove FG assignment
        while((address[i]!=key) && (i<7)){i++;}
        if(address[i]!=key){reset();}
        address[i]=0x00;
        PORTE = i;
    }
}

// Reset PIC, if undesired command found
void reset(void){
}

```

## C. Verilog

```
`timescale 1ns / 1ps
/////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    23:04:20 11/08/2009
// Design Name:
// Module Name:    midisynth_ibj1
// Project Name:
// Target Devices:
// Tool versions:
// Description: Overall top level module for synth with 8 independant wave
//              generators summed together for final output. This module expects
//              a 20 MHz input clock.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
/////////////////////////////////////////////////////////////////
module midisynth_ibj1(
    input [6:0] key,
    input [6:0] velocity,
    input keyon,
    input en,
    input [2:0] addr,
    input clk,
    input reset,
    output [15:0] audioout
);

    wire [7:0] keyen;
    wire enedge;

    wire [9:0] addr0;
    wire [9:0] addr1;
    wire [9:0] addr2;
    wire [9:0] addr3;
    wire [9:0] addr4;
    wire [9:0] addr5;
    wire [9:0] addr6;
    wire [9:0] addr7;

    wire [15:0] data0;
    wire [15:0] data1;
    wire [15:0] data2;
    wire [15:0] data3;
    wire [15:0] data4;
    wire [15:0] data5;
    wire [15:0] data6;
    wire [15:0] data7;

    wire [15:0] audioout0;
    wire [15:0] audioout1;
    wire [15:0] audioout2;
    wire [15:0] audioout3;
    wire [15:0] audioout4;
    wire [15:0] audioout5;
    wire [15:0] audioout6;
    wire [15:0] audioout7;

    wire [18:0] audiosum;

    // CLOCK DIVISION
```



```

easydivider #(8) divider(clk, reset, slowclk); //NON-DEBUGGING

//wire slowclk;           // DEBUGGING ONLY
//assign slowclk = clk;   // DEBUGGING ONLY

// ENABLE EDGE DETECTING, WAVE GENERATOR ADDRESS DECODING

rising edgedetector leveltopulse(en, slowclk, reset, enedge);
threetoeight addrtokeyen(addr, keyen);

// EIGHT WAVE GENERATORS

onewavegenenv wavegen0(slowclk, reset, key, velocity, keyon, (keyen[0] & enedge), audioout0,
addr0, data0);
onewavegenenv wavegen1(slowclk, reset, key, velocity, keyon, (keyen[1] & enedge), audioout1,
addr1, data1);
onewavegenenv wavegen2(slowclk, reset, key, velocity, keyon, (keyen[2] & enedge), audioout2,
addr2, data2);
onewavegenenv wavegen3(slowclk, reset, key, velocity, keyon, (keyen[3] & enedge), audioout3,
addr3, data3);
onewavegenenv wavegen4(slowclk, reset, key, velocity, keyon, (keyen[4] & enedge), audioout4,
addr4, data4);
onewavegenenv wavegen5(slowclk, reset, key, velocity, keyon, (keyen[5] & enedge), audioout5,
addr5, data5);
onewavegenenv wavegen6(slowclk, reset, key, velocity, keyon, (keyen[6] & enedge), audioout6,
addr6, data6);
onewavegenenv wavegen7(slowclk, reset, key, velocity, keyon, (keyen[7] & enedge), audioout7,
addr7, data7);

// MEMORY AND 8 PORT CONVERSION

eightportrom #(10, 16) romaccessor(addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7,
data0, data1, data2, data3, data4, data5, data6, data7,
clk, reset);

// FINAL OUTPUTS

assign audiosum = (audioout0 + audioout1 + audioout2 + audioout3 +
audioout4 + audioout5 + audioout6 + audioout7);
assign audioout = audiosum[18:3]; // divide by 8

endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    01:11:03 09/18/2009
// Design Name:
// Module Name:    easydivider
// Project Name:
// Target Devices:
// Tool versions:
// Description: Less customizable but much simpler divider. Divides clock speed by 32768
//               to get about 60 Hz from 20 MHz clock input.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module easydivider #(parameter BITS = 8) (
    input fastclk,
    input reset,
    output slowclk
);

    reg [BITS-1:0] count;
    wire [BITS-1:0] countplus1;

    always @ (posedge fastclk, posedge reset)
        if (reset) count <= 0;
        else      count <= countplus1; // on clock edge, counter increments to value already
determined by CL

    assign countplus1 = count + 1;      // figure out next count with combinational logic
    assign slowclk = count[BITS-1];    // MSB is used as output slower clock
endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    14:44:56 11/21/2009
// Design Name:
// Module Name:    8portrom
// Project Name:
// Target Devices:
// Tool versions:
// Description: 8 port rom from single port rom (time interleaving)
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module eightportrom #(parameter addrbits = 10, databits = 16)
(
    input [addrbits-1:0] addr0,
    input [addrbits-1:0] addr1,
    input [addrbits-1:0] addr2,
    input [addrbits-1:0] addr3,
    input [addrbits-1:0] addr4,
    input [addrbits-1:0] addr5,
    input [addrbits-1:0] addr6,
    input [addrbits-1:0] addr7,
    output [databits-1:0] data0,
    output [databits-1:0] data1,
    output [databits-1:0] data2,
    output [databits-1:0] data3,
    output [databits-1:0] data4,
    output [databits-1:0] data5,
    output [databits-1:0] data6,
    output [databits-1:0] data7,
    input fastclk,
    input reset
);

    wire [addrbits-1:0] romaddr;
    wire [databits-1:0] romdata;

    reg [2:0] port; // select signal for which port is being used

    wire [addrbits-1:0] addr0reg;
    wire [addrbits-1:0] addr1reg;
    wire [addrbits-1:0] addr2reg;
    wire [addrbits-1:0] addr3reg;
    wire [addrbits-1:0] addr4reg;
    wire [addrbits-1:0] addr5reg;
    wire [addrbits-1:0] addr6reg;
    wire [addrbits-1:0] addr7reg;

    wire [addrbits-1:0] addr0reg2;
    wire [addrbits-1:0] addr1reg2;
    wire [addrbits-1:0] addr2reg2;
    wire [addrbits-1:0] addr3reg2;
    wire [addrbits-1:0] addr4reg2;
    wire [addrbits-1:0] addr5reg2;
    wire [addrbits-1:0] addr6reg2;
    wire [addrbits-1:0] addr7reg2;

    // Synchronizer

    flopenr #(addrbits) sync0(fastclk, reset, 1'b1, addr0, addr0reg);
    flopenr #(addrbits) sync1(fastclk, reset, 1'b1, addr1, addr1reg);

```

```

flopnr #(addrbits) sync2(fastclk, reset, 1'b1, addr2, addr2reg);
flopnr #(addrbits) sync3(fastclk, reset, 1'b1, addr3, addr3reg);
flopnr #(addrbits) sync4(fastclk, reset, 1'b1, addr4, addr4reg);
flopnr #(addrbits) sync5(fastclk, reset, 1'b1, addr5, addr5reg);
flopnr #(addrbits) sync6(fastclk, reset, 1'b1, addr6, addr6reg);
flopnr #(addrbits) sync7(fastclk, reset, 1'b1, addr7, addr7reg);

flopnr #(addrbits) sync20(fastclk, reset, 1'b1, addr0reg, addr0reg2);
flopnr #(addrbits) sync21(fastclk, reset, 1'b1, addr1reg, addr1reg2);
flopnr #(addrbits) sync22(fastclk, reset, 1'b1, addr2reg, addr2reg2);
flopnr #(addrbits) sync23(fastclk, reset, 1'b1, addr3reg, addr3reg2);
flopnr #(addrbits) sync24(fastclk, reset, 1'b1, addr4reg, addr4reg2);
flopnr #(addrbits) sync25(fastclk, reset, 1'b1, addr5reg, addr5reg2);
flopnr #(addrbits) sync26(fastclk, reset, 1'b1, addr6reg, addr6reg2);
flopnr #(addrbits) sync27(fastclk, reset, 1'b1, addr7reg, addr7reg2);

// mux to select which address is being fed to the rom
mux8 #(addrbits) inputmux(port,
                                addr0reg2, addr1reg2, addr2reg2, addr3reg2,
                                addr4reg2, addr5reg2, addr6reg2, addr7reg2,
                                romaddr);

reg [7:0] regenable;

always @(*)
begin
    case (port) // 3:8 decoder for enabling the 8 storage registers
        3'h0: regenable <= 8'b0000_0001;
        3'h1: regenable <= 8'b0000_0010;
        3'h2: regenable <= 8'b0000_0100;
        3'h3: regenable <= 8'b0000_1000;
        3'h4: regenable <= 8'b0001_0000;
        3'h5: regenable <= 8'b0010_0000;
        3'h6: regenable <= 8'b0100_0000;
        3'h7: regenable <= 8'b1000_0000;
    endcase
end

flopnr #(databits) d0flop(fastclk, reset, regenable[0], romdata, data0);
flopnr #(databits) d1flop(fastclk, reset, regenable[1], romdata, data1);
flopnr #(databits) d2flop(fastclk, reset, regenable[2], romdata, data2);
flopnr #(databits) d3flop(fastclk, reset, regenable[3], romdata, data3);
flopnr #(databits) d4flop(fastclk, reset, regenable[4], romdata, data4);
flopnr #(databits) d5flop(fastclk, reset, regenable[5], romdata, data5);
flopnr #(databits) d6flop(fastclk, reset, regenable[6], romdata, data6);
flopnr #(databits) d7flop(fastclk, reset, regenable[7], romdata, data7);

wavformmemrom #(10) wvmem(romaddr,romdata);

always @(posedge fastclk)
    if (reset)
        port <= 3'b000;
    else
        port <= (port+1);
endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    23:46:30 12/01/2009
// Design Name:
// Module Name:    envelopegen
// Project Name:
// Target Devices:
// Tool versions:
// Description:    Overall envelope generator. Expects an enable that is high for
//                one clock cycle only to internally update on/off status. Note
//                on/off should not put through a register external to this
//                module.
//
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module envelopegen #(parameter TIMEBITS = 20) (
    input [16:0] A,
    input [16:0] D,
    input [16:0] R,
    input [TIMEBITS-1:0] Atime,
    input [TIMEBITS-1:0] Dtime,
    input onoff,
    input en,
    input clk,
    input reset,
    output reg [15:0] envelope
);

    wire [16:0] envinc;
    wire [15:0] nextenv;
    wire [TIMEBITS-1:0] currttime;
    wire onoffreg; // register whether note is on or off, using enable to update

    // time elapsed since last note turned on
    notetimerreg #(TIMEBITS) timereg(en, onoff, clk, reset, currttime);

    // update note on/off when en received.
    flopenr #(1) onoffflop(clk, reset, en, onoff, onoffreg);

    // get increment to feed to adder
    envincrement incrementalcalc(A, D, R, Atime, Dtime, currttime, onoffreg, envinc);

    // saturation adder for next envelope value
    satadder #(17) envadder(envinc, {1'b0, envelope}, nextenv);

    always @(posedge clk)
    begin
        if (reset)
            envelope <= 0; // reset time to 0;
        else if (en && onoff) // new note on and enable high for update
            envelope <= 0; // also should reset time to 0;
        else
            envelope <= nextenv;
    end
endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    23:28:15 12/01/2009
// Design Name:
// Module Name:    envincrement
// Project Name:
// Target Devices:
// Tool versions:
// Description:    ADSR envelope, outputs the increment to the current envelope
//                  register value.
//
//                  Note that the increments are 17 bits since they are 2's
//                  complement numbers that still allow the full 16 bit unsigned
//                  range.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module envincrement #(parameter TIMEBITS = 20) (
    input [16:0] A,           // increment in attack stage
    input [16:0] D,           // increment in delay stage (negative 2's comp number)
    input [16:0] R,           // increment in rel. stage (negative 2's comp number)
    input [TIMEBITS-1:0] Atime, // end time of the attack section
    input [TIMEBITS-1:0] Dtime, // end time of the delay section
    input [TIMEBITS-1:0] timein, // time since note start
    input onoff,
    output reg [16:0] envinc
);

    always @(*)
    begin
        if (~onoff)           // in release stage (note is off)
            envinc <= R;
        else if (timein < Atime) // in attack stage (note on, less than Atime)
            envinc <= A;
        else if (timein < Dtime) // in delay stage (note on, Atime < time < Dtime)
            envinc <= D;
        else                   // in sustain stage (note on, time > Dtime)
            envinc <= 0;       // no change in sustain area
    end

endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    02:21:20 09/24/2009
// Design Name:
// Module Name:    flopenr
// Project Name:
// Target Devices:
// Tool versions:
// Description: Simple parameterized enabled, resettable, register.
//                (Asynchronous reset)
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module flopenr
    #(parameter width = 4)
    ( input clk,
      input reset,
      input en,
      input [width-1:0] d,
      output reg [width-1:0] q
    );

    always @(posedge clk, posedge reset)
        if (reset) q <= 0;
        else if (en) q <= d;

endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    23:15:18 11/08/2009
// Design Name:
// Module Name:    keytomemincrement
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module keytomemincrement(
    input [6:0] key,
    output reg [23:0] meminc
);

    always @(*)
        case (key)
            7'b0000000: meminc = 24'b000000000000000011011011;
            7'b0000001: meminc = 24'b000000000000000011101000;
            7'b0000010: meminc = 24'b000000000000000011110110;
            7'b0000011: meminc = 24'b0000000000000000100000100;
            // middle left out to save space.
            7'b1111011: meminc = 24'b000001000001001111111000;
            7'b1111100: meminc = 24'b000001000101001000001011;
            7'b1111101: meminc = 24'b000001001001001111010000;
            7'b1111110: meminc = 24'b000001001101100101111110;
            7'b1111111: meminc = 24'b000001010010001101010001;
            default: meminc = 24'h000000;
        endcase
endmodule

```



```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    15:47:46 11/21/2009
// Design Name:
// Module Name:    mux8
// Project Name:
// Target Devices:
// Tool versions:
// Description: 8 input mux to use for 8 port rom
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module mux8 #(parameter WIDTH = 8) (
    input [2:0] select,
    input [WIDTH-1:0] d0,
    input [WIDTH-1:0] d1,
    input [WIDTH-1:0] d2,
    input [WIDTH-1:0] d3,
    input [WIDTH-1:0] d4,
    input [WIDTH-1:0] d5,
    input [WIDTH-1:0] d6,
    input [WIDTH-1:0] d7,
    output reg [WIDTH-1:0] y
);

    always @(*)
    begin
        case (select)
            3'h0: y <= d0;
            3'h1: y <= d1;
            3'h2: y <= d2;
            3'h3: y <= d3;
            3'h4: y <= d4;
            3'h5: y <= d5;
            3'h6: y <= d6;
            3'h7: y <= d7;
            default: y <= d0;
        endcase
    end
endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    23:03:43 12/01/2009
// Design Name:
// Module Name:    notetimerreg
// Project Name:
// Target Devices:
// Tool versions:
// Description: keeps track of samples elapsed since beginning of note. If
//               overflows, will "get stuck" at the max value the time register
//               can hold.
//
//               20 bits gives 13 seconds of timing at 78125 Hz.
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module notetimerreg #(parameter BITS = 20) (
    input en,
    input onoff,
    input clk,
    input reset,
    output reg [BITS-1:0] timeout
);

    wire [BITS-1:0] nexttime;

    wire [BITS:0] wideone;

    genvar i;

    // 1 of the proper bus width
    generate
        for (i = 0; i < (BITS+1); i = i+1) begin:forloop
            if (i == 0)
                assign wideone[i] = 1;
            else
                assign wideone[i] = 0;
        end
    endgenerate

    wire [BITS:0] timeoutwide;
    assign timeoutwide = {1'b0, timeout};

    // use saturation adder
    satadder #(BITS+1) nexttimeadder(timeoutwide, wideone, nexttime);

    always @(posedge clk)
    begin
        if (reset)
            timeout <= 0;                // reset time to 0;
        else if (en && onoff)
            timeout <= 0;                // new note on and enable high for update
            // also should reset time to 0;
        else
            timeout <= nexttime;
    end

endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    23:09:44 11/08/2009
// Design Name:
// Module Name:    onewavegen
// Project Name:
// Target Devices:
// Tool versions:
// Description: Wave generator with more complex envelope added.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module onewavegenenv(
    input clk,
    input reset,
    input [6:0] key,
    input [6:0] velocity,
    input on,
    input en,
    output [15:0] audioout,
    output [9:0] wvfmmaddr,
    input [15:0] wvfmmdata
);

    // Envelope variables:
    wire [16:0] A;
    wire [16:0] D;
    wire [16:0] R;
    wire [19:0] Atime;
    wire [19:0] Dtime;

// // DEBUGGING CONSTANTS:
/* assign A = 17'h02222;
   assign D = 17'h1FFF00;
   assign R = 17'h1FFFE;
   assign Atime = 20'hFFFFFFE;
   assign Dtime = 20'hFFFFFF;*/

// RUNTIME CONSTANTS: 419ms, 419ms to 50%, 100 ms release
assign A = 17'h00002;
assign D = 17'h1FFFF;
assign R = 17'h1FFFC;
assign Atime = 20'h07FFF;
assign Dtime = 20'h0FFFF;

wire [23:0] meminc;

wire [15:0] envelope;

wire [6:0] regvel;           // stored velocity from velocity flop
wire [6:0] regkey;         // stored key from key flop

// Flops store velocity and key, update on enable only.

flopnr #(7) velocityflop(clk, reset, en, velocity, regvel);
flopnr #(7) keyflop(clk, reset, en, key, regkey);

// Waveform memory increment is based on frequency, using a

```

```

// ROM look up table.

keytomemincrement meminclut(regkey, meminc);
wavformaddr addrreg(clk, reset, meminc, wvfmdr);

// shift waveform down
wire signed [16:0] shifteddownwvfm;
assign shifteddownwvfm = {1'b0, wvfmdr} + 17'h18001; // move down by -7FFF

// then multiply
wire signed [16:0] sgnenvelope;
wire signed [7:0] sgnregvel;
wire signed [33:0] envmultdata;
wire signed [25:0] velmultdata;
wire signed [16:0] envmultdivided;
wire signed [16:0] velmultdivided;

assign sgnenvelope = {1'b0, envelope};
assign sgnregvel = {1'b0, regvel};
assign envmultdata = shifteddownwvfm * sgnenvelope;
assign envmultdivided = envmultdata[33:16];
assign velmultdata = envmultdivided * sgnregvel;
assign velmultdivided = velmultdata[23:7];

// shift waveform up and take take must significant bits
wire [16:0] shiftedupwvfm;
wire [15:0] audiooutunreg;
assign shiftedupwvfm = velmultdivided + 17'h07FFF; // move up
assign audiooutunreg = (envelope == 16'b0 || regvel == 7'b0 ||
    wvfmdr == 16'h7FFF) ? 16'h7FFF : shiftedupwvfm[15:0];

flopenr #(16) audiooutreg(clk, reset, 1'b1, audiooutunreg, audioout);

envelopegen #(20) envgen(A, D, R, Atime, Dtime, on, en, clk, reset, envelope);

endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    12:21:09 09/24/2009
// Design Name:
// Module Name:    risingedgedetector
// Project Name:
// Target Devices:
// Tool versions:
// Description: Uses two flops in series as a sort of shift register with logic to
//              detect if a rising edge has occurred (the last sample was low, but
//              the newest sample is high). Can be used as a level to pulse
//              converter.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module risingedgedetector(
    input a,
    input clk,
    input reset,
    output risingedgedetected
);

    wire newer, older;          // newer is output of first flop

    flopr flop1(clk, reset, a, newer);
    flopr flop2(clk, reset, newer, older);

    assign risingedgedetected = ~older & newer;
endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    22:11:11 12/01/2009
// Design Name:
// Module Name:    satadder
// Project Name:
// Target Devices:
// Tool versions:
// Description: Implements a specific sort of "saturation" type arithmetic.
//                Specifically, the inputs are assumed to be two's complement
//                signed numbers. The output is constrained to the range from
//                0 to  $2^{(N-1)} - 1$ , where N is the number of input bits. If
//                the result (Y) would be negative, the output is 0 instead. If
//                the result would overflow in the positive direction, the result
//                is instead  $2^{(N-1)}$ .
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module satadder #(parameter WIDTH = 17) (
    input [WIDTH-1:0] A,
    input [WIDTH-1:0] B,
    output reg [WIDTH-2:0] Y
);

    wire [WIDTH-1:0] plainsum;
    wire ovfl; // result overflow
    wire neg; // result negative
    wire [WIDTH-2:0] allones;

    genvar i;

    // all ones, of width matching the output width.
    generate
        for (i = 0; i < (WIDTH-1); i = i+1) begin:forloop
            assign allones[i] = 1;
        end
    endgenerate

    assign plainsum = A + B;
    assign neg = plainsum[WIDTH-1]; // if MSB high, result is negative

    // ref'd Wakerly's Digital Design p. 491 for:
    assign ovfl = (A[WIDTH-1] == B[WIDTH-1]) && (plainsum[WIDTH-1] != A[WIDTH-1]);

    always @(*)
    begin
        if (ovfl) // overflow
            Y <= allones[WIDTH-2:0]; // all high if overflow
        else if (neg) // didn't overflow, but is negative
            Y <= 0; // make output 0 instead
        else
            Y <= plainsum[WIDTH-2:0]; // normal output, without  $-2^N$  MSB
    end

endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    17:59:31 11/21/2009
// Design Name:
// Module Name:    threetoeight
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module threetoeight(
    input [2:0] a,
    output reg [7:0] d
);

    always @(*)
    begin
        case (a)
            3'h0: d <= 8'b0000_0001;
            3'h1: d <= 8'b0000_0010;
            3'h2: d <= 8'b0000_0100;
            3'h3: d <= 8'b0000_1000;
            3'h4: d <= 8'b0001_0000;
            3'h5: d <= 8'b0010_0000;
            3'h6: d <= 8'b0100_0000;
            3'h7: d <= 8'b1000_0000;
        endcase
    end

endmodule

```

```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    23:18:22 11/08/2009
// Design Name:
// Module Name:    waveformaddr
// Project Name:
// Target Devices:
// Tool versions:
// Description: Register for storing current waveform address within an individual
//                note generator.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module waveformaddr(
    input clk,
    input reset,
    input [23:0] meminc,
    output [9:0] addr
);

    reg [23:0] addrreg;

    always @( posedge clk)
    begin
        if (reset) addrreg <= 24'b0;
        else addrreg <= (meminc + addrreg);
    end

    assign addr = addrreg[23:14];
endmodule

```



```

`timescale 1ns / 1ps
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Ian Bullock (ibullock@hmc.edu)
//
// Create Date:    23:11:15 11/08/2009
// Design Name:
// Module Name:    waveformmem
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
module waveformmemrom # (parameter N = 10)
    (input [N-1:0] addr,
     output [15:0] data
    );

    wire [15:0] mem [2**N-1:0];

    assign mem[10'b0000000000] = 16'h7fff;
    assign mem[10'b0000000001] = 16'h8647;
    assign mem[10'b0000000010] = 16'h8c8b;
    assign mem[10'b0000000011] = 16'h92c7;
    assign mem[10'b0000000100] = 16'h98f8;
    // middle skipped for conciseness
    assign mem[10'b1111111100] = 16'h6706;
    assign mem[10'b1111111101] = 16'h6d37;
    assign mem[10'b1111111110] = 16'h7373;
    assign mem[10'b1111111111] = 16'h79b7;

    assign data = mem[addr];
endmodule

```