A Simple Drum Loop Recorder and Sequencer

Dmitriy Yakovlev and Chris Koo

December 11, 2009

Abstract

Many electronic drum kits allow the operator to load use various nonstandard drum samples to produce a unique beat. In a similar vein, loop stations allow the operator to record a short sound sequence and play it back repeatedly for use as a backing sound to other music. These two ideas were combined together to create a multi-channel drum loop recorder and sequencer as a final project. This drum machine plays six different drum sounds corresponding to six different buttons. It can record and replay looped sequences of the drum sounds from three different channels. The PIC handles button presses and records the sequences while relaying information to the FPGA. The FPGA accepts the given input triggers and synthesizes 12-bit digital combinations of different beeps and whirrs that act as drum sounds, which it outputs in parallel fashion to a DAC and an amplification stage, which plays the amplified sound through a speaker. Analogous, albeit more complex, devices include the Korg Micromoog and the Roland drum synth series produced in the 1970s.

1 Introduction

Our drum machine is a multi-channel recorder and sequencer specifically tailored to recording, storing and playing back multiple looped drum beats. It is thus the synthesis of several different ideas: a multi-track recorder, a loop station, and a drum machine.

The drum machine has two modes: a record mode and a play mode. In record mode, the user picks one of 99 storage channels via channel selection buttons and presses the start/stop button to begin recording a sequence. Then, the user can put in any drum sequence they choose by pressing buttons corresponding to different drum samples at the times they want those samples to be played back. The sound of the corresponding sample is played as the user presses each drum button. By pressing the start/stop button again, the recording is terminated and can be played back by entering play mode.

In play mode, the user picks a channel in similar fashion to record mode, and presses the start/stop button to start looped playback of the sequence stored in that channel. At any time, they can press the start/stop button again to stop playback, and the channel up/down buttons to move to a different channel, which will also stop playback until the start/stop button is pressed again.

We have implemented this machine in hardware using the microcontroller resources available on the Harrisboard, namely the PIC and the FPGA, as well as various hardware and integrated circuits external to the Harrisboard. By implementing this design in hardware, we have gained a deep appreciation for the art of Digital Sound Processing and efficient data storage both on the PIC and the FPGA.

2 Design

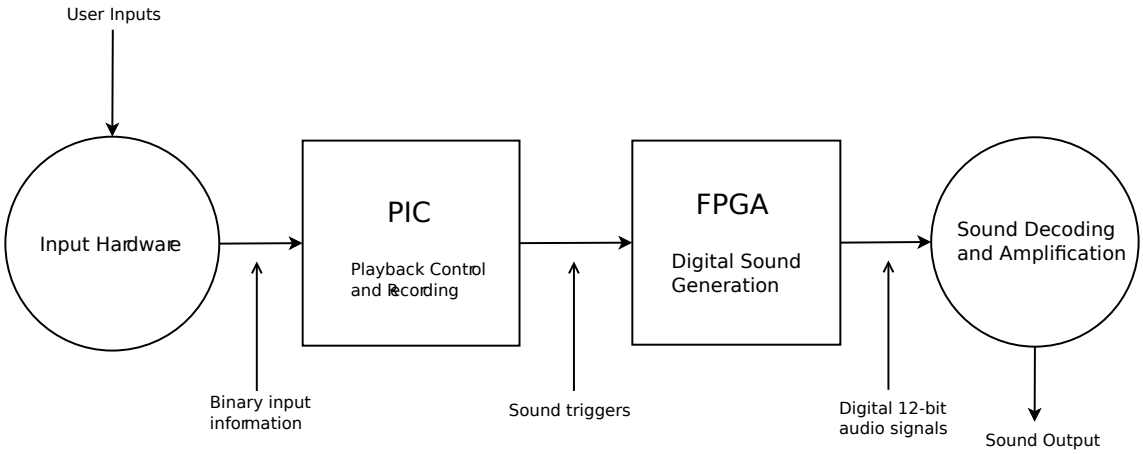


Figure 1: Overall information flow through the design.

There are four major components in our system: the input hardware, the PIC, the FPGA, and the output hardware. The overall system design is dictated by the responsibilities of each major component, as well as the one-way direction flow throughout the entire system. The input hardware consists of an array of buttons that lead to weak-low input pins on the PIC. Six of these buttons control the drum sounds, and the rest control channels and playback as well as switching between record and playback modes.

The PIC functions mainly as an input and playback processor as well as storage for the recorded sequences. It interacts intimately with the input hardware, monitoring the input lines to detect button presses and interact accordingly. The FPGA functions primarily as a sound generator, controlled by inputs from either the PIC or external stimulus. It interacts intricately with the output hardware, putting out more than 44 thousand samples per second to a digital-to-analog converter, which sends the converted analog voltage to an amplifier and eventually to a pair of speakers.

2.1 Input Stage Design

The input hardware design is relatively simple. It consists of nine normally-open pushbuttons and a SPST switch. All nine input channels are held low, which is achieved by 44KOhm pulldown resistors on the input lines. Six of these nine inputs correspond to the previously mentioned drum buttons. Two more are a "channel up" and a "channel down" button, and are used to go between sequence channels on the PIC. One momentary button is used as the Start/Stop button, and controls the starting and stopping of playback or recording. The PIC's mode is toggled between playback and recording by the SPST switch. Two LEDs on the input side of things are actually output status indicators. One lights up red to signify entry into record mode, and the other lights up green to signify that playback or recording is currently occurring.

To get a picture of the overall input circuit, consult Appendix 4.3.1.

2.2 PIC Design

The PIC functions as an input and output controller. It takes in external interrupts from the start/stop button and channel up/down buttons. When the interrupts occur, the interrupts are handled based on the flags that are raised. When the start button is pressed, an interrupt event occurs and sets a global variable, Start, high. When the toggle is in play mode, the main function will recognize the Start variable as high and start accessing the memory in order to relay the necessary drum sequence information to the FPGA.

The information the PIC stores for each drum hit is the DrumID and timer, which respectively correspond to the identification number of the drum buttons pressed and the amount of time that has passed since the last button press. The drum IDs in the global structure array will be sent to the FPGA until the stop button is pressed or the pointer reaches the end of the sequence within the channel, at which point it will loop back to the beginning of that sequence and play it again.

When the play versus record toggle is set to the record mode, an interrupt will occur which will cause the pointer to go to the beginning of the sequence in the channel and wait for the start button to be pressed. Once the start button is pressed, the PIC starts accessing the data storage and stores the timer and drum identification values each time a button is pressed or multiple buttons are pressed simultaneously. Recording will continue until the limit of the data allocated in the channel is reached or the stop button is pressed.

Interrupts will occur when the channel up or down buttons are pressed. When the channel buttons are pressed, regardless of the current processes in the PIC, the PIC will drop all current work and switch to the channels by moving the pointer around and clearing the timer and start variable.

The PIC's operation can be fairly simply explained in the form of a finite state machine. See Appendix 4.3.2 for that representation.

2.3 FPGA Design

The FPGA has been designed to support one main functionality requirement: to synthesize and output drum sound samples based on stored data and received controls. Currently, it synthesizes various beep sound samples, although drum-like noises can currently be implemented without a lot of work.

The FPGA modules have been written to support a fairly classical DSP arrangement. A 44100Hz output sample clock has been defined, and controls the output sample speed of the entire circuit. The rest of the FPGA is dedicated to six synthesizer channels which are added together at the end to produce a combined sound sample.

Each synthesizer channel (referred to as synth) contains three things: a sample generator, a numerically-controlled ADSR filter, and a multiplier with playback controls. Each channel is usually arranged like the block diagram in Figure 2.

In this structure, the sample generator actually contains the waveform that will be played back, in an internal ROM. At every 44.1KHz clock cycle, the sample generator moves to the next 12-bit sample, rolling back to the start when it gets to the end. The samples stored in the generator are in a 12-bit signed format and can generally be anything stored as an uncompressed sequence of amplitudes. For the purposes of testing and experimental sound generation, we loaded our sample generators with sine wave samples generated by the Perl script attached at Appendix 4.4.3.

The ADSR numerically-controlled filter is an envelope filter with configurable rates of Attack, Decay, Sustain, and Release. It is generally used to simulate a lot of different audio events, because the ADSR pattern is highly representative of impact-based noises - it starts with a jump up to a very high amplitude, decays, and stays at the same amplitude coefficient for a while before finally decaying to zero. It is implemented using a finite state machine, which controls which phase the NCF is in at any given point

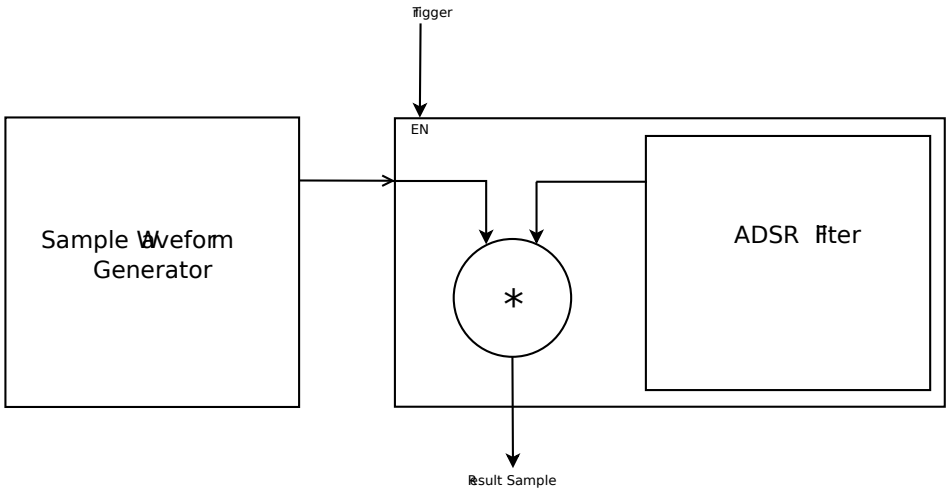


Figure 2: Block-level Channel Structure.

in time, as well as handling trigger/mute input control. At every clock cycle, the NCF returns a signed 12-bit coefficient to the synth module. When this coefficient decreases to zero, the sample stops playing.

The main synth component, seen surrounding the ADSR NCF, contains a multiplier and some combinational logic to control the states of its internal FSM, which controls whether a sample should be played back from the channel or not. The way this logic works necessitates that a single trigger pulse is sent on the input trigger line. This will start sample playback, which will continue until a mute signal is sent or the ADSR filter's coefficient decreases to zero. At every time step, the synth takes a sample from the generator, a coefficient from the ADSR, and multiplies them to get the resultant sample, which is then put out. This works to scale the amplitude of the generated sound because both the sample and the coefficient are returned as signed 12-bit numbers, and the SG samples are centered around zero, which makes multiplying by a strictly-positive coefficient produce mathematically correct waveforms.

2.4 Output Stage Design

The output stage of the design is very simple. The FPGA takes the 12-bit digital sound output from pins P10-P24 and feeds it directly into a Maxim MAX507 12-bit DAC. This DAC has two latches on the input, presumably for capturing fluctuating digital input, but it has been configured to leave both transparent, which allows the use of an internal register to buffer the output inside the FPGA instead of doing it in hardware. The DAC is configured to output a voltage from 0 to 5V depending on the binary input, and this output voltage represents the output signal if the input is varied at a constant update frequency.

The output analog signal is sent through a 6 KHz cutoff low-pass filter and into one half of a 12W stereo power amp called the TDA7263. The low-pass filter is necessary to reduce the amount of digital and analog hum due to power supply noise and FPGA output gate instability. This amp outputs the sound into a pair of 40W 5'1/4" speakers designed for automotive audio. The TDA7263 is configured for variable gain to allow at least some control of volume level. Consult Appendix 4.3.1 for the amplifier hookup configuration used.

3 Results

The project currently successfully functions as a multi-channel drum sound recorder and sequencer. However, due to implementation difficulties, some parts of the specification had to be worked around to produce a functional result.

The PIC has all of the functionality stated in the design; however, the access to the ROM was difficult in C. In one version of the PIC code, the ROM is accessed and data is stored in the ROM, but the ROM pointer of the array would not move. The PIC would create an array large enough to fit 99 channels, but the pointer could not access the array other than the zero point of the array. The PIC had to be reverted to storing sequences in the RAM, which could only store up to three channels due to its relatively small size.

The FPGA has all the stated functionality as well, but we did not have time to find nice drum clips to distill into repeatable waveforms to store in the sound generator ROMs. As a result, it currently outputs various frequencies of sine wave, with various envelopes applied to them to produce a range of sounds from a chirp to a low, deep grunt. None of those are technically drum noises, but they were sufficient to test the device's sound synthesis capabilities.

4 Additional Information and Appendices

4.1 Parts List

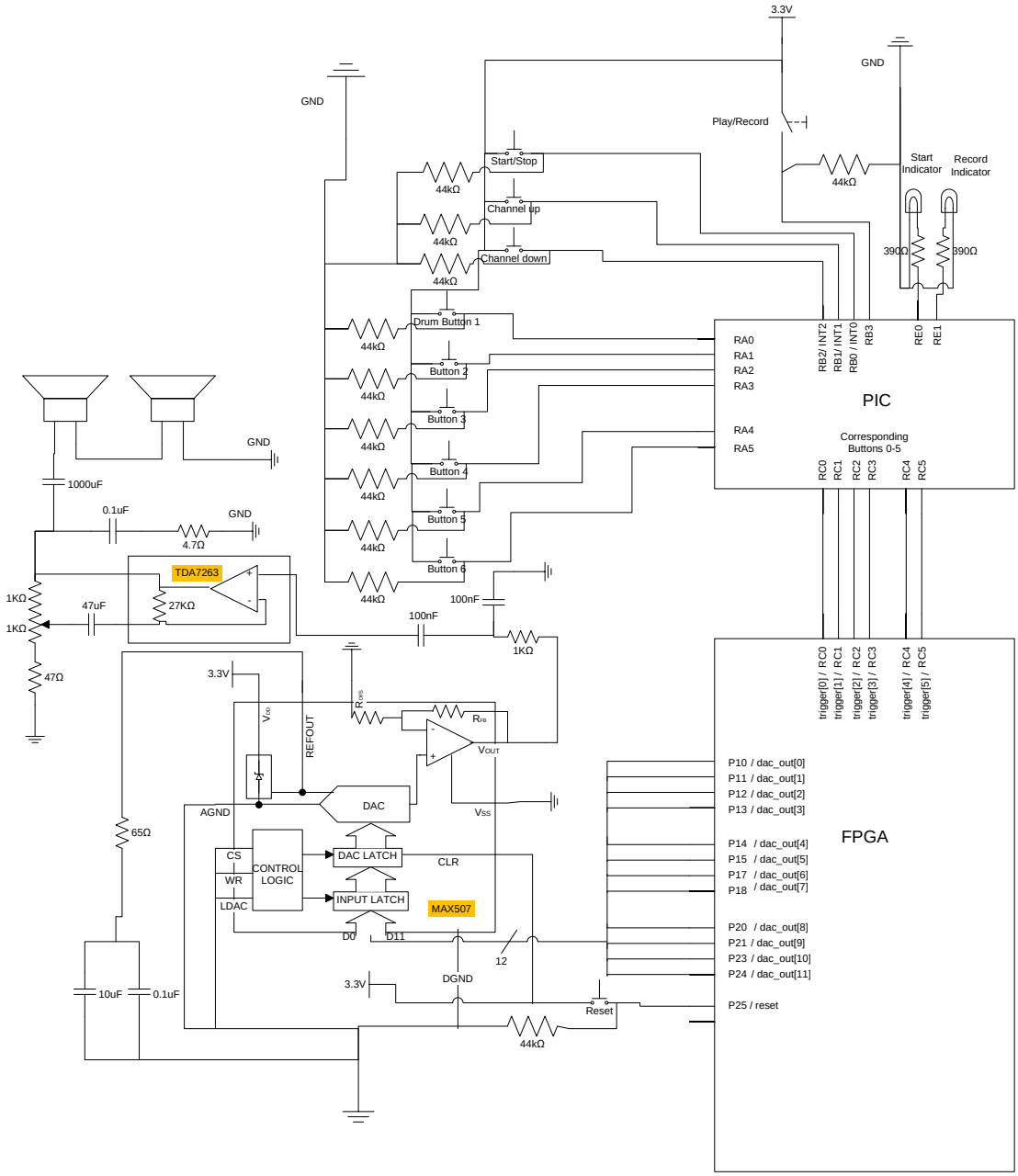
Quantity	Part	Comment	Price (total)
10	PBNO pushbutton	From stockroom	-
2	MAXIM MAX507	12-bit parallel DAC, from stockroom	-
1	STMicroelectronics TDA7263	497-3958-5-ND Dual op-amp, Digikey	$3.04
4	47 uF capacitor	P5156-ND Digikey	$0.44
4	1000 uF capacitor	P13458-ND Digikey	$2.32
2	Sony XPLOD 5.25 speaker	On loan from Dmitriys supplies	-

4.2 References

- `http://www.maxim-ic.com/quick_view2.cfm/qv_pk/1543` - DAC information
- `http://www.datasheetcatalog.com/datasheets_pdf/T/D/A/7/TDA7263.shtml` - Power Amp information
- `http://en.wikipedia.org/wiki/Low_pass_filter` - Low Pass Filter Reference
- `http://www.dspguide.com/` - DSP information

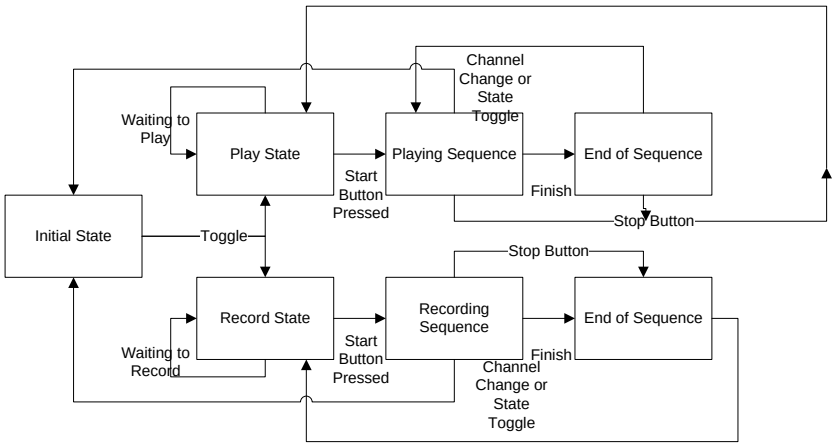
4.3 Schematics

4.3.1 Breadboard

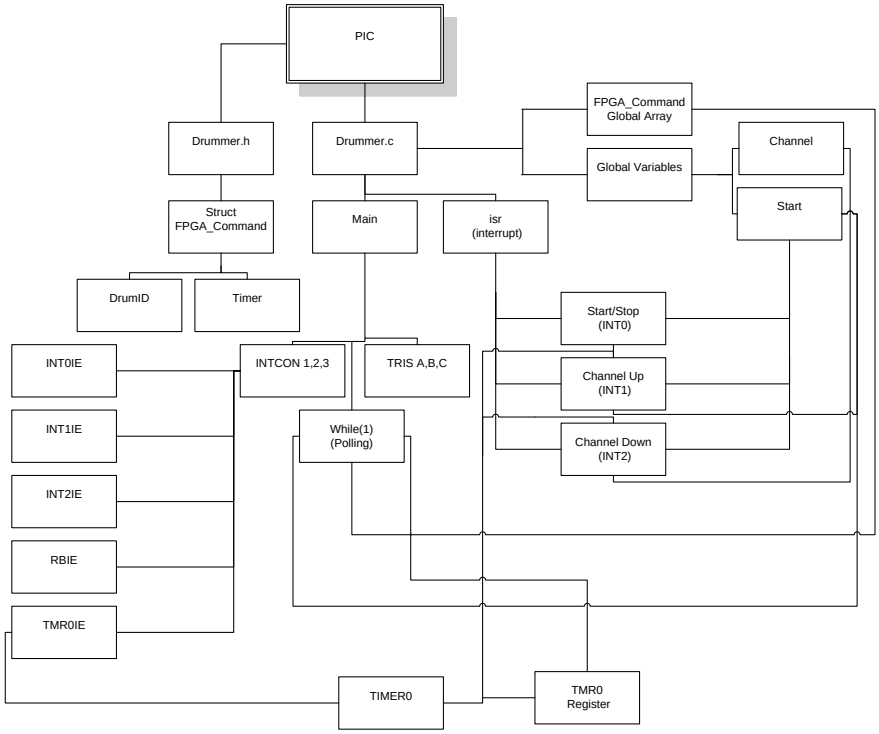


4.3.2 PIC

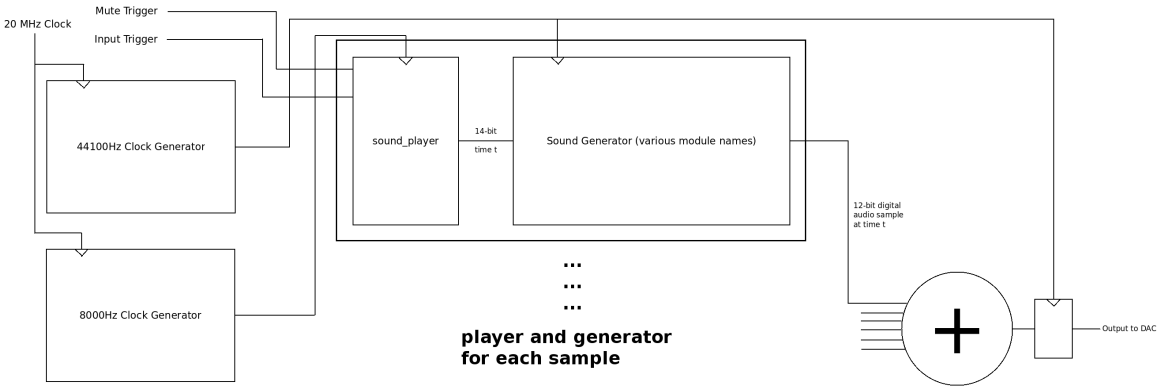
Finite State Machine Representation of Control Flow



Routines


PIC
Drummer.h
Drummer.c
FPGA_Command
Global Array
Global Variables
Channel
Start
Struct
FPGA_Command
Main
isr
(interrupt)
DrumID
Timer
Start/Stop
(INT0)
INT0IE
INTCON 1,2,3
TRIS A,B,C
Channel Up
(INT1)
INT1IE
While(1)
(Polling)
Channel Down
(INT2)
INT2IE
RBIE
TMR0IE
TIMER0
TMR0
Register

4.3.3 FPGA


Mute Trigger
20 MHz Clock
Input Trigger
44100Hz Clock Generator
8000Hz Clock Generator
sound_player
14-bit
time t
Sound Generator (various module names)
12-bit digital
audio sample
at time t
...
...
...
player and generator
for each sample
Output to DAC

4.4 Code Listing

4.4.1 PIC C Code

Listing 1: drummer.h

```
/*
        E155 Final Project - Drummer

        PIC C18 header code.

        Chris Koo bckoo9@gmail.com
        Dmitriy Yakovlev dyakovlev@hmc.edu

*/

// data types used for storage
struct fpga_command
{
    unsigned int timer;
    unsigned char DrumID;
};

// Function Prototypes
void main(void);
void isr(void);
```

Listing 2: drummer.c

```
/*
    E155 Final Project - Drummer

    PIC C18 code. Handles sequence recording, storage, and playback control.

    Chris Koo bckoo9@gmail.com
    Dmitriy Yakovlev dyakovlev@hmc.edu

    input pins
        RA0-5: drums
        RB0: start/stop PBNO
        RB1: channel up PBNO
        RB2: channel down PBNO
        RB3: play/record switch - 1 = Play, 0 = Record

    output pins
        RC0-5: FPGA drum control
        RD0: on in record mode, off in play mode
        RD1: on when playing, off when stopped
*/

#include <p18f4520.h>
#include <delays.h>
#include <timers.h>
#include <usart.h>
#include "drummer.h"
```

```
// global vars and storage
far struct fpga_command Command[75];
far struct fpga_command *pointer;
int Start, Channel;

// The #pragma tells the compiler to start a code section, named
// high_vector at the program memory address of 0x08. This is
// the interrupt vector address.
#pragma code high_vector = 0x08
void high_interrupt(void) {
_asm    GOTO isr    _endasm
}

// Now start the main code section.
#pragma code
void main(void)
{
    unsigned char last_buttons;
    unsigned char playback;

    // note - use LATx instead of PORTx for writing to output pins
    LATA = 0x00; // clear PORTA
    LATB = 0x00; // clear PORTB
    LATC = 0x00; // clear PORTC
    LATE = 0x00;

    ADCON0bits.ADON = 0; // ADC off
    ADCON1 = 0x0F; // digital IO, not analog reads

    // config IO directions
    TRISA = 0b00111111;     // A <-
    TRISB = 0b00001111;     // B[3:0] <-
    TRISC = 0b00000000;     // C ->
    TRISE = 0b00000000;     // E ->

    // config internal and external interrupts
    INTCON =  0b11110000;   //Init interrupts for global, peripheral, and INT0
    INTCON2 = 0b11110100;   //Init priority of RB interrupts, allow interrupt
    INTCON3 = 0b11011000;   //Set priority for INT1+INT2, enable external ints

    T0CON = 0b10000111;     //Set up Timer 0 for everything

    pointer = &Command[0]; // current playback/recording position
    Start = 0;          // 1 when start has been pressed
    Channel = 0;        // current channel #
    last_buttons = 0;   // used for remembering button presses
    playback = PORTBbits.RB3;

    // Do the following forever
    while(1) {
        // if we just switched between playback and record,
        // stop and reset pointer to beginning of channel
        if (playback != PORTBbits.RB3) {
```

```
    Delay10KTCYx(1); // 10,000 cycles at 20 mhz ~ 8ms
    PORTC = 0;
    pointer = &Command[0] + Channel*25;
    Start = 0;
    playback = PORTBbits.RB3;
}

// if we should be doing anything (if !play, nothing happens)
if (Start) {
    LATEbits.LATE0 = 1;

    // If we are in playback mode
    if(PORTBbits.RB3){
        LATEbits.LATE1 = 0; // playback mode

        // if we're at the end of the channel, loop back to start
        if(pointer > (&Command[0]+Channel*25+24)) {
            LATC = 0; // zero output
            pointer = &Command[0]+Channel*25;
        }
        // if all is OK and we should advance in the sequence, do so
        else if(ReadTimer0() >= (*pointer).timer) {
            LATC = (*pointer).DrumID; // which drums to start playing
            pointer++; // advance to the next command
            WriteTimer0(0); // reset the timer

            // hold input high for a while
            Delay10KTCYx(1); // 10,000 cycles at 20 mhz ~ 8ms
        }
        // if we haven't reached the time for the next sequence, wait
        else {
            LATC = 0;
        }
    }

    // if we're in record mode
    else {
        LATEbits.LATE1 = 1; // record mode

        // if we have run out of channel to record into
        if(pointer > (&Command[0]+Channel*25+24)){
            LATC = 0;
            Start = 0;
            pointer = &Command[0]+Channel*25;
        }
        else {
            // still recording... is a button pressed?

            if(PORTA==0 && last_buttons==0); // no buttons pressed
            else if(last_buttons==0 && PORTA!=0){ // buttons pressed
                Delay10KTCYx(1); // 10,000 cycles at 20 mhz ~ 8ms
                LATC=PORTA;
                last_buttons=PORTA;
                (*pointer).DrumID=PORTA;
```

```
                        (*pointer).timer= ReadTimer0();

                        pointer++;
                        WriteTimer0(0);
                    }
                    else if(PORTA==0 && last_buttons!=0){ // buttons pressed
                        last_buttons=0;
                        LATC = 0;
                    }
                }
            }
        }
        else {
            LATEbits.LATE0 = 0; // off when not playing

            if(PORTBbits.RB3) LATEbits.LATE1 = 0; // playback mode
            else              LATEbits.LATE1 = 1; // record mode

            LATC = PORTA;
            Delay10KTCYx(1); // 10,000 cycles at 20 mhz ~ 8ms
        }
    }
}

// The #pragma lets compiler know isr() is an interrupt
// handler.
#pragma interrupt isr
void isr(void)
{
    Delay10KTCYx(1); // 10,000 cycles at 20 mhz ~ 8ms

    if(INTCONbits.INT0IF) {   //Interrupt start stop

        INTCONbits.INT0IF=0;
        WriteTimer0(0);
        if(Start) {
            Start=0;
            pointer=&Command[0]+Channel*25;
        }
        else {
            Start=1;

            // if we're in record mode, clear the channel
            if(!PORTBbits.RB3)
            {
                while(pointer <= &Command[0]+Channel*25+24)
                {
                    (*pointer).timer=0;
                    (*pointer).DrumID=0;
                    pointer++;
                }
                pointer = &Command[0]+Channel*25;
            }
        }
```

```
    }
    else if(INTCON3bits.INT1IF) {    //Interrupt channel up
        INTCON3bits.INT1IF=0;
        if(Channel==2)
            Channel=0;
        else
            Channel++;

        Start=0;
        pointer=&Command[0]+Channel*24;
        LATD = Channel;
    }
    else if(INTCON3bits.INT2IF) {     //Interrupt channel down
        INTCON3bits.INT2IF=0;
        if(Channel==0)
            Channel=2;
        else
            Channel--;

        Start=0;
        pointer=&Command[0]+Channel*24;
        LATD = Channel;
    }
    else if(INTCONbits.TMR0IF) {      //Interrupt timer overflow
        INTCONbits.TMR0IF=0;
        if(!PORTBbits.RB3){
            (*pointer).timer=0xFFFF;
            (*pointer).DrumID=0x00;
            pointer++;
            WriteTimer0(0);
        }
        else
        {
            WriteTimer0(0);
            pointer++;
        }
    }
}
```

4.4.2 FPGA Code

Listing 3: controller.v

```
/*
    E155 Final Project - Drummer

    Xilinx Spartan3 FPGA Code. Handles sound synthesis and 44.1KHz output.

    Chris Koo bckoo9@gmail.com
    Dmitriy Yakovlev dyakovlev@hmc.edu

    input pins
        clk - P124 Clock
        reset - P25 Reset
        triggers - P82-P87 RC input from PIC

    output pins
        dac_out - P10-P24 DAC Sample output
        leds - P97-P105 LED Bargraph output
*/

`timescale 1ns / 1ps

/*  Controller Module

    Instantiates a number of synthesizer channels,
    adds their output at every 44100Hz cycle and puts it out on
    dac_out
*/
module controller(
    input clk,                      // 20 MHz FPGA system clock
    input reset,                    // routed to synth channels
    input [5:0] triggers,       // drum inputs (several can fire at once)
    output reg [11:0] dac_out,  // 12-bit parallel DAC output
    output [7:0] leds           // LED debug output
    );

    assign leds = {2'b00,triggers};

    wire [14:0] dac_in; // dac data buffer
    wire sample_clk;

    // slowed-down audio output clock (44100.0 Hz)
    clockdiv output_clk(clk, 32'd220500, sample_clk);
    // note: the input # is duration of half cycle

    // channel 0
    wire [11:0] signal_0;
    wire [11:0] output_0;
    synth #(24'h000500, 24'h000300, 24'h000100, 24'h000400) c0(sample_clk,
        reset, triggers[0], signal_0, output_0);
    sine_220Hz sig0(sample_clk, signal_0);

    // channel 1
```

```
    wire [11:0] signal_1;
    wire [11:0] output_1;
    synth #(24'h000500, 24'h000300, 24'h000100, 24'h000400) c1(sample_clk,
        reset, triggers[1], signal_1, output_1);
    sine_120Hz sig1(sample_clk, signal_1);

    // channel 2
    wire [11:0] signal_2;
    wire [11:0] output_2;
    synth #(24'h000500, 24'h000300, 24'h000100, 24'h000400) c2(sample_clk,
        reset, triggers[2], signal_2, output_2);
    sine_80Hz sig2(sample_clk, signal_2);

    // channel 3
    wire [11:0] signal_3;
    wire [11:0] output_3;
    synth #(24'h000500, 24'h000300, 24'h000100, 24'h000400) c3(sample_clk,
        reset, triggers[3], signal_3, output_3);
    sine_330Hz sig3(sample_clk, signal_3);

    // channel 4
    wire [11:0] signal_4;
    wire [11:0] output_4;
    synth #(24'h000500, 24'h000300, 24'h000100, 24'h000400) c4(sample_clk,
        reset, triggers[4], signal_4, output_4);
    sine_400Hz sig4(sample_clk, signal_4);

    // channel 5
    wire [11:0] signal_5;
    wire [11:0] output_5;
    synth #(24'h000500, 24'h000300, 24'h000100, 24'h000400) c5(sample_clk,
        reset, triggers[5], signal_5, output_5);
    sine_60Hz sig5(sample_clk, signal_5);

    // combines the individual sample signals
    signal_combiner sc1(output_0, output_1, output_2,
                        output_3, output_4, output_5,
                        dac_in);

    // flip-flop gates output to DAC
    always @ ( posedge sample_clk )
        dac_out <= dac_in[14:3];

endmodule

/*  Synth channel module

    Controls one drum. Trigger starts playback, mute stops it before it stops
    automatically by reaching ADSR endpoint
*/
module synth(
    input sample_clk,    // 44100Hz sample rate clock
    // controls
    input mute,      // stop playing
```

```
input trigger,  // start playing (or restart)
// sample ROM access port
input [11:0] sample_in,
// resulting sample output (0x7FF when not playing)
output reg [11:0] sample_out,
);

// ADSR envelope params (passed from initializing module)
parameter a_rate = 0;
parameter d_rate = 0;
parameter s_rate = 0;
parameter r_rate = 0;

// force treat sample_in as signed
wire signed [11:0] sample_ins;
assign sample_ins = sample_in;

wire signed [11:0] coeff; // return wire for adsr coefficient
wire signed [23:0] wreg; // working reg for adjusted sample calculation
wire        [11:0] wout; // output reg for adjusted sample calculation

// state machine stuff
reg [1:0] state = 2'b00;
reg [1:0] next = 2'b00;
parameter Trig = 2'b01;
parameter Playing = 2'b10;
parameter Stopped = 2'b00;

wire done; // adsr signal

// this module gives back envelope multiplier coefficient
adsr_envelope #(a_rate, d_rate, s_rate, r_rate)
    adsr1(sample_clk, state[0], coeff, done);

// multiply! (sample gets >>>1 to get coeff to line up right)
assign wreg = (sample_ins>>>1) * coeff;

// snips top 12 bits and converts signed -> unsigned
assign wout = wreg[23:12] + 12'h7FF;

// state register
always @ (posedge sample_clk)
begin
    state <= next;
    case (state)
    Playing:
        sample_out <= wout;
    Trig:
        sample_out <= wout;
    default: // Stopped
        sample_out <= 12'h7FF;
    endcase
end
```

```
    // next state logic
    always @ ( * )
    begin
        case (state)
        Trig:
            next <= Playing;
        Playing:
            if (trigger) next <= Trig;
            else if (mute || done) next <= Stopped;
            else next <= Playing;
        Stopped:
            if (trigger) next <= Trig;
            else next <= Stopped;
        default:
            next <= Stopped;
        endcase
    end

endmodule

/*  ADSR envelope NCF module

    applies ADSR envelope to input data.
*/
module adsr_envelope(
    input sample_clk,
    input play_ctl,          // pulse indicates start/restart playback
    output [11:0] coeff,     // multiplication coefficient
    output idlestate
    );

    // coeff output range: 0x000 to 0x7FF

    // ADSR envelope params (get declared at module init)
    parameter a_rate = 0;
    parameter d_rate = 0;
    parameter s_rate = 0;
    parameter r_rate = 0;

    // ADSR turning points (hardcoded here)
    parameter [23:0] attack_turn = 24'h7FF000;  // 2*ampltiude
    parameter [23:0] decay_turn = 24'h3FF000;   // 1*amplitude
    parameter [23:0] sustain_turn = 24'h1FF000; // 1/2*amplitude

    reg signed [23:0] coeff_reg; // accumulator for the coefficient
    assign coeff = coeff_reg[23:12]; // snip and output top 12

    // predefined add/subtract values
    wire signed [23:0] a_inc;
    assign a_inc = coeff_reg + a_rate;
    wire signed [23:0] d_dec;
    assign d_dec = coeff_reg - d_rate;
    wire signed [23:0] s_dec;
    assign s_dec = coeff_reg - s_rate;
```

```
wire signed [23:0] r_dec;
assign r_dec = coeff_reg - r_rate;

// state machine that controls stage of the adsr
reg [2:0] state = 3'b100;
reg [2:0] next = 3'b100;

parameter Idle    = 3'b101;
parameter Trigger = 3'b100;
parameter Attack  = 3'b000;
parameter Decay   = 3'b001;
parameter Sustain = 3'b010;
parameter Release = 3'b011;

// state register
always @ (posedge sample_clk)
begin
    state <= next;
    case (state)
    Idle:
        coeff_reg <= 24'h2FF000; // *1
    Trigger:
        coeff_reg <= 24'h2FF000; // resets coeff
    Attack:
        coeff_reg <= a_inc;
    Decay:
        coeff_reg <= d_dec;
    Sustain:
        coeff_reg <= s_dec;
    Release:
        coeff_reg <= r_dec;
    default:
        coeff_reg <= 24'h3FF000; // *1
    endcase
end

// next state logic
always @ ( * )
begin
    case (state)
    Idle:
        if (play_ctl) next <= Trigger;// retrigger
        else next <= Idle;
    Trigger:
        next <= Attack;
    Attack:
        if (play_ctl) next <= Trigger;// retrigger
        else if (coeff_reg > attack_turn) next <= Decay;
        else next <= Attack;
    Decay:
        if (play_ctl) next <= Trigger;// retrigger
        else if (coeff_reg < decay_turn) next <= Sustain;
        else next <= Decay;
    Sustain:
```

```
            if (play_ctl) next <= Trigger;// retrigger
            else if (coeff_reg < sustain_turn) next <= Release;
            else next <= Sustain;
        Release:
            if (play_ctl) next <= Trigger;// retrigger
            else if (coeff_reg < 0) next <= Idle;
            else next <= Release;
        default:
            next <= Idle;
        endcase
    end

    // high when the adsr is idling
    assign idlestate = (state == 3'b101);

endmodule

// adds six signed 12-bit values
// note - gives 15-bit result instead of overflowing or clipping
module signal_combiner(
    input [11:0] sample0,
    input [11:0] sample1,
    input [11:0] sample2,
    input [11:0] sample3,
    input [11:0] sample4,
    input [11:0] sample5,
    output [14:0] sum
    );

    // note - consider replacing with 13-bit clipping adder
    assign sum = (sample0 + sample1 + sample2 + sample3 + sample4 + sample5);

endmodule

// clock divider, outputs square signals of any frequency less than 10 MHz
// note - frequency is an integer determining the quantity of 10ths of hertz..s,
//        so to get 5 kHz, put 50000 (5000.0 Hz) into frequency
module clockdiv(
    input clk,
    input [31:0] frequency,
    output freq_out
    );

    reg [31:0] count = 0;    // holds current cycle count

    always @ (posedge clk)
    begin
        if (count + frequency < 200000000) count = count + frequency;
        else count = (count + frequency) - 200000000;
    end

    assign freq_out = (count >= 100000000);

endmodule
```

Listing 4: Sample Sine Wave Generator

```
/*
        E155 Final Project - Drummer

        Verilog ROM and counter modules for sine wave generation.

        Chris Koo bckoo9@gmail.com
        Dmitriy Yakovlev dyakovlev@hmc.edu

*/
`timescale 1ns / 1ps
/*
    ROM module with  12-bit samples of a 80Hz sine wave taken at 44100Hz
*/

module sine_80Hz(
    input clk,
    output [11:0] data    // signed 12-bit sample
    );

    reg [9:0] address = 0; // 10-bit address

    sine_80Hz_data data(address, data);

    always @ (posedge clk)
        if (address == 10'b1000100111) address = 0;
        else address = address + 1;

endmodule

module sine_80Hz_data(
    input [9:0] address,
    output reg [11:0] data
    );

    always @ ( * )
    begin
        case(address)
            10'b0000000000: data = 12'b000000000000;
            10'b0000000001: data = 12'b000000010110;
            10'b0000000010: data = 12'b000000101101;
            10'b0000000100: data = 12'b000001011100;
                            ...
            10'b1000100100: data = 12'b111110110110;
            10'b1000100101: data = 12'b111111001101;
            10'b1000100110: data = 12'b111111100100;
            10'b1000100111: data = 12'b111111111100;
            default:
                data = 12'h800;
        endcase
    end
endmodule
```

4.4.3 Perl Code

Listing 5: gen.pl

```
#!/usr/bin/perl

# E155 Final Project - Drummer
# Perl code that generates Verilog sine wave NCO modules
# Dmitriy Yakovlev dyakovlev@hmc.edu

# takes frequency, sampling frequency, output file name as inputs
# e.g. gen.pl 440 44100 A4.v

# prints out loaded Verilog ROM

die("bad inputs") if $#ARGV != 2;

$f = @ARGV[0];
$sf = @ARGV[1];
$file = @ARGV[2];

print "\nsine frequency: $f Hz\nsampling at: $sf Hz";
print "\ntarget file: $file";

open (TARGET, ">>$file") or die $!;

# setup

my $pi = 3.14159265358979;

sub d2b {
    return unpack("B32", pack("N", shift));
}

# generate beginning of module
$pre = ''`timescale 1ns / 1ps
/*
    ROM module with  12-bit samples of a ' .$f. 'Hz sine wave taken at '.$sf.'Hz
*/

module sine_' . $f . 'Hz(
    input clk,
    output [11:0] data   // signed 12-bit sample
    );

    reg [9:0] address = 0; // 10-bit address

    sine_' . $f . 'Hz_data data(address, data);

    always @(posedge clk)
        if (address == 10\'b' . substr(d2b(int($sf/$f)), -10) . ') address = 0;
        else address = address + 1;

endmodule
```

```
module␣sine_' . $f . 'Hz_data(
␣␣␣␣input␣[9:0]␣address,
␣␣␣␣output␣reg␣[11:0]␣data
␣␣␣␣);

␣␣␣␣always␣@␣(␣*␣)
␣␣␣␣begin
␣␣␣␣␣␣␣␣case(address)
';

# and put it into the file
print TARGET $pre;

# gen cases and write into file
foreach $sample (0 .. ($sf/$f)) {

    $data = sin(2*$pi*$sample*$f / $sf);

    # scale up to 2^12
    $data = int($data * 2048) * 0.999;

    # convert data to 12-bit binary, snip
    $bdata = substr d2b($data), -12;

    # convert address to 10-bit binary, snip (shouldn't overflow)
    $badr = substr d2b($sample), -10;

    print TARGET "␣␣␣␣␣␣␣␣␣␣␣␣10'b$badr:␣data␣=␣12'b$bdata;\n";
}

# generate end of module. standard output is 1000000000000, which is "0"
$post = "␣␣␣␣␣␣␣␣␣␣␣␣default:
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣data␣=␣12'h800;
␣␣␣␣␣␣␣␣endcase
␣␣␣␣end
endmodule
";

# and put it in the file
print TARGET $post;
close (TARGET) or die $!;
print "\n";
```