HARVEY MUDD COLLEGE

Monophonic Digital MIDI-Controlled Synthesizer with Bitcrusher Effect

E155 Final Report

Leo Altmann and Madeleine Ong

12/11/2009

A PIC microcontroller and a Xilinx FPGA were used to build a monophonic digital synthesizer. The MIDI protocol was used for control. Pitch is controlled by the keys on a keyboard, and a rotary control knob is used to change parameters on a bitcrusher effect. The output is a sinusoid with varying degrees of amplitude resolution. The final deliverable met all the original specifications. Notable design challenges were the MIDI receiver circuit and post-bitcrusher amplitude correction.

Introduction

Project Overview

The goal of our project was to build a MIDI-controlled monophonic digital synthesizer. MIDI signals are sent from an M-Audio O2 keyboard to a PIC for decoding. The PIC interprets the signals, determining note on / off status, frequency, and control parameters for a bitcrusher (digital quantization distortion) effect. These signals will be passed to the FPGA, which will generate a sine wave of corresponding frequency, modify the signal with the bitcrusher circuit, and output the result to a D/A converter.

Figure 1: Block Diagram of System

As the bitcrush effect becomes more pronounced, the original sinusoid output looks closer to a series of step function, as shown in *Figure 2*, adding a sheen to the tone.

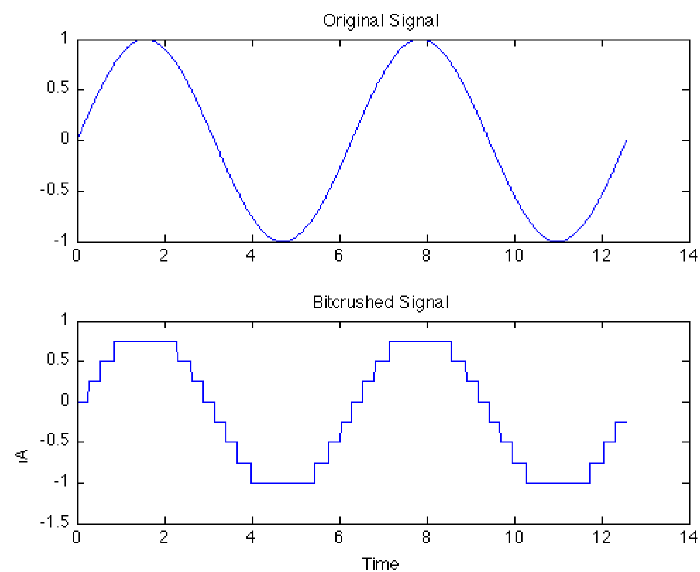


Figure 2: Sample bitcrushed signal in comparison to original sinusoid output

System Partition

The PIC was chosen to receive MIDI messages in order to take advantage of the capabilities of its build-in USART serial transceiver. (Note: please refer to *Appendix A* for more details about the MIDI protocol.) The MIDI protocol specification calls for an optocoupler circuit on receivers, to separate the voltage and current used in transmission from those used as input to the processing circuit and to eliminate ground loops between noise-sensitive audio devices. Such a circuit was implemented from the standard schematic provided in the specification, using a 6N138 optocoupler. MIDI messages are decoded on the PIC, producing note number, note on/off, and control message values. The PIC steps through values of an angle theta, representing relative position within one period of a sine wave, and sends the values to a sine lookup table on the FPGA when note on is asserted. Changing the rate at which the wave is stepped through produces a different pitch.

The PIC interprets valid control bytes from a rotary control on the keyboard to produce a divisor for use in the bitcrusher routine, and passes this value to the FPGA alongside the phase value. Major components on the FPGA are a sine lookup table, a divider and a multiplier, all synthesized with Xilinx CoreGen. Sine wave outputs from the lookup table are passed to the bitcrusher, where they are divided by the divisor from the PIC, rounded down, and multiplied by the divisor to make the amplitude closer to its original value. Bitcrushed values are passed to the D/A converter, and the output signal is amplified by an LM386 chip-amp driving a generic 8-ohm speaker. A full schematic of the breadboarded circuits can be found in *Appendix E*.

New Hardware

This project used a digital to analog converter (DAC), model AD558K DACPORT®. The DAC converts the digital output from the FPGA into an analog sinusoid for a speaker. The AD558 takes in 8 channels of

data on pints DB0 to DB7 in parallel, giving it 8-bits of resolution with a numerical range of 0 to 255. In this way, a fraction of the reference voltage, V_{dd} can be output proportional to the digital input. The pin configuration of the DIP AD558 is shown in *Figure 3* below.

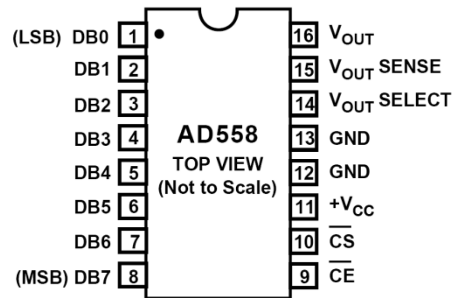


Figure 3: AD558 Pin Configuration (DIP) [5]

The AD558 was powered by a +5V source, and outputs between 0 and +2.56 V due to pin-strapping of V_{OUT} to V_{OUT}SENSE and V_{OUT}SELECT, shown in *Figure 4*.

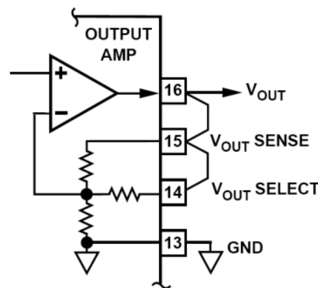


Figure 4: Pin-strapping of AD558 to select output voltage range. [5]

The DAC works by latching in the input values to the internal registers when chip enable bar ($\overline{\mathrm{CE}}$) or chip select bar ($\overline{\mathrm{CS}}$) is set high and converting them into an analog voltage that is sent put out of V_{OUT} when they are set low, making the DAC transparent. When both are set low, the chip is transparent and the data is continuously converted.

Optocoupler Circuit

The MIDI protocol specification calls for an optocoupler (or opto-isolator) circuit to be used as a buffer on input terminals. An optocoupler uses an LED, powered by the input signal, to switch a phototransistor and control current flow on the output. In the MIDI specification, an optocoupler is used to isolate the current and voltage used to transmit between devices from more sensitive devices, such as ICs, and to reduce audible distortion from ground loops between devices.

A schematic of the standard MIDI optocoupler circuitry from the protocol specification is shown in *Figure 5*. The input is connected through a 270-ohm resistor to limit the current to 20mA from the 5.4V transmission source, and a 1N914 diode was used to divert back-current from the optocoupler's internal LED. Toggling the input will use the LED to control the phototransistors, thereby toggling the output between the high-potential 5V source and the ground sink. An 0.1uF smoothing capacitor was added between power and ground to reduce noise on the optocoupler output.

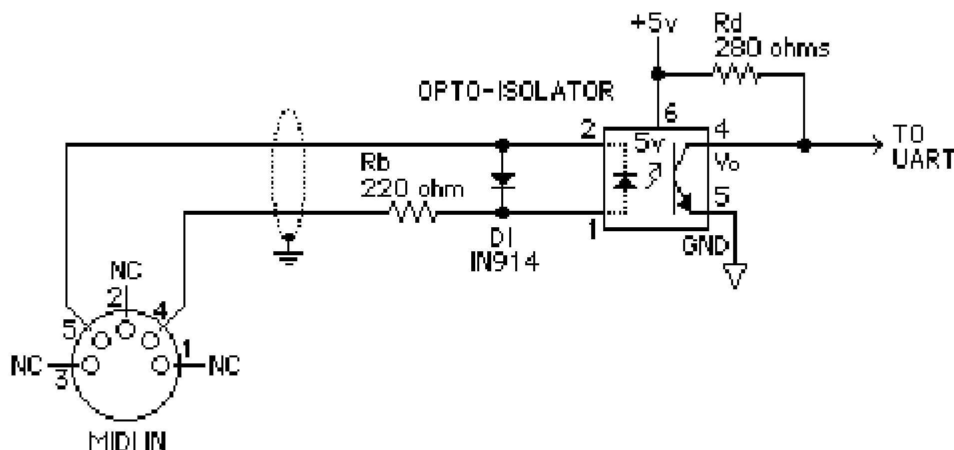


Figure 5: Optocoupler circuit for MIDI input, as defined in the protocol specification. [1]

Microcontroller Design

The PIC handles MIDI reception and decoding, keeps track of wave position and phase step, and passes note on / data ready control signals, theta and divisor values to the FPGA. MIDI signals were received by

the USART on pin RC7. The USART transmitter was disabled, and the rest of PORTC was used to output the bitcrusher divisor. Only RB1 and RB0 were used from PORTB, for ready and note on respectively. The 13-bit phase value was passed to the FPGA using PORTA (RA4:RA0) for the 5 most significant bits and PORTD for the 8 least significant bits. PORTE was not used.

Timer0 was configured to trigger interrupts at a rate of 44.1kHz. When an interrupt occurs, the PIC clears the data ready output flag, computes the new theta from the current value and the step size for the current frequency, writes the phase value to PORTA and PORTD, writes the crush divisor to PORTC, and re-asserts the ready flag. Polling for MIDI data was chosen instead of using a second interrupt to ensure that generating audio samples would take first priority, thereby reducing discontinuities in the output signal.

The PIC code is split in to four main sections: initialization, MIDI reception, MIDI decoding and sample generation. The initialization section defines variables, sets the tri-state buffers for each port, configures USART, timer and interrupt control registers, and initializes outputs to their note-off state. The MIDI reception code is a finite state machine that polls for new MIDI bytes and assembles them into complete MIDI messages. The FSM also tracks "running status" messages, a feature of the protocol where multiple data values of the same type are sent in rapid succession after a single status byte. It stores the most recent status byte and passes it to the decoder again if the first byte of a new message is data (value is < 128).

A subroutine to decode incoming MIDI messages is called by the receiver FSM when it as assembled a complete message. If the status byte denotes a note on event, the value of the phase step per sample is changed according to the note number received. Phase steps are stored in a program memory data table, indexed by note number. The values were computed beforehand using the Python script in

Appendix B. Note on messages with zero velocity are interpreted as note off messages. When a control message is received, if it is from the correct control knob and non-zero, it is set as the new bitcrusher divisor. Sample generation code is run every time timer0 issues an interrupt. The data ready flag is set low, a new phase value is computed from the previous phase and the phase step value, the divisor and new phase value are written to their respective data ports, and the ready flag is reasserted.

FPGA Implementation

The FPGA was used to create the sine wave and bitcrush the values according to the dial control on the keyboard. This was achieved through the use of three Xilinx CoreGen modules [4]: SinCos, Divider and Multiplier. These modules generated a sine value from relative phase position, reduced the resolution of the value, and scaled up the divided value, respectively. The timing of the output data was controlled by a ready signal from the PIC, representing when the received data bytes were valid. A general block diagram is shown in *Figure 6*.

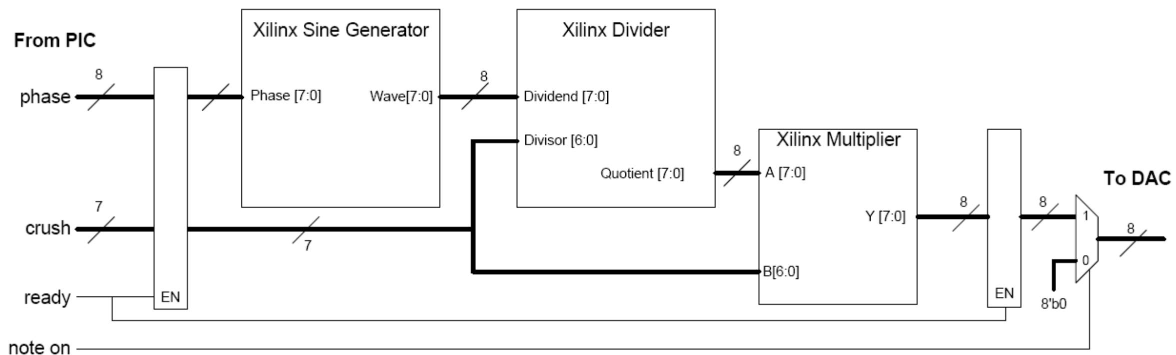


Figure 6: Block diagram of FPGA functions

The first core is SinCos v5.0, a sine function generator. It takes in the input theta, representing the fractional position within one period of a sine wave, and outputs the value of the sine function at the given position. The input theta was configured to be thirteen bits wide, in order to maximize the pitch

resolution of the sine wave. The output was 8 bits wide, to make the eventual output compatible with the 8-bit DAC.

The second core used is Divider v5.0, and is used to produce the bitcrusher effect. The 8-bit output from the sine generator is divided by the divisor (crush amount) sent from the PIC. This is proportional to the dial on the keyboard. The resulting quotient is also 8-bits, with the remainder ignored.

This value was then scaled back up using the Multiplier v11.2 core. This is done to maximize the oscillation magnitude so the signal can be audibly heard when eventually passed through the DAC. The divisor was used as the scaling factor in order to recreate a value of similar magnitude to the original value. Since the lost resolution cannot be recreated, this results in quantization noise that changes the character of the sound. Output values from the bitcrusher are passed to the D/A converter when they become available.

These values are forwarded conditionally, based on the input ready and note on signal. The ready signal indicates the data is valid when it goes low. At the time the signal goes low, the DAC controls for $\overline{\mathrm{CE}}$ and $\overline{\mathrm{CS}}$ are similarly set low to allow the valid output data to go through. This data is either the crushed value from the sine generator or silence, based on whether note on is high or low, respectively. This prevents metastable or incorrect data from being outputted.

Results

The final deliverable met all the goals set out in the project proposal. The schematics of the breadboarded circuits and block diagrams are located in the Appendices. MIDI note and control messages are properly received and decoded, samples are generated at 44.1kHz producing a sine wave of appropriate frequency when a key is pressed, and the bitcrusher introduces quantization noise to the

signal as expected. An additional feature was also added: if two keys are held at once, when one is released the other note resumes playing.

MIDI reception proved to be more difficult to implement than was originally anticipated. Optocouplers are sensitive devices, as was discovered when one was destroyed early in development and went unnoticed for a period of time. The output from the circuit was also inverted from what we expected, generating garbage data and framing errors on every byte until the issue was located. An inverter IC was used initially, but removed later once the USART configuration for inverted input was found. Even after the optocoupler circuit was producing clean, accurate signals, receiving MIDI bytes with the USART produced inconsistent results with the device in low-speed mode. Consistent results were achieved using the device in high-speed mode with the 16-bit internal timer. Final configuration values are shown in *Table 1*.

Table 1: PIC configuration register values used for MIDI reception. 'x' denotes don't-care.

Register	**Value**	**(Bits) Description**
`RCSTA`	`0b10010000`	(7) Enable USART (6) 8-bit messages (5) x (4) Enable receiver (3) Disable address detection (2:0) Clear interrupt flags
`TXSTA`	`0b01000100`	(7) x (6) 8-bit messages (5) Disable transmitter (4) Asynch. mode (3) x (2) High-speed mode (1:0) Clear interrupt flags
`BAUDCON`	`0b00101000`	(7:6) Clear status bits (5) RX signal is inverted (4) TX idle low (3) Use 16-bit timer (2) x (1) Continuously sample input (0) Disable auto baud-rate detection
`SPBRG`	`0b10011111`	Calculated timer offset for 31250 baud with above settings (see PIC18LF4520 datasheet)

MIDI reception was further complicated by the presence of unexpected messages in the datastream. The protocol supports System Real-Time messages, which are meant to be transmitted at regular intervals for timing purposes. In our case, the keyboard was transmitting a form of heartbeat byte called an Active Sensing message every 300ms, at times even in the middle of other messages. The solution implemented adds an extra loop around the poll and receive byte routine that discards bad bytes and continues polling until a valid byte is received. It was also found to be advantageous to

separate the MIDI reception and decoding processes into two different routines. This simplified the receiver state machine, reducing the code to approximately half its original size, and eliminated several large, complex conditional structures.

Several timing and resolution issues were also encountered that were not related to serial reception. Original plans called for an 8-bit phase value to drive the sine generator on the FPGA, but additional precision was added later on. This allowed the synthesizer to achieve the desired sample rate without rounding errors drastically altering the pitch at lower frequencies. The final implementation used a 13-bit phase value, transmitted over PORTD and most of PORTA. Additionally, the sample timing was tuned so that the pitch of notes played matched standard frequencies. Using an oscilloscope, a timer offset value was experimentally determined to ensure proper sample rate timing despite the delay from interrupt code execution.

The FPGA successfully created a sine wave with resolution proportional to the keyboard dial input. The distortion was visible on an oscilloscope and produced the predicted audio effect. There was issues with the amplitude, as output from the ADC have varying amplitudes in the oscillation based on the size of the divider. Whenever the divider was a multiple of two, the signal had the maximum amplitude. Otherwise, the amplitude decreased as the divider increased between these powers of two. This was due to the method of reintroducing amplitude. The multiplier did not increase the amplitude enough, but it was sufficient to be heard over the speaker.

There were significant differences in our original design of the FPGA modules compared to what was eventually used. Initially we had only two Xilinx CoreGens, with a priority encoder instead of the multiplier to reintroduce the correct amplitude. This led to problems as not enough amplitude was reintroduced with a simple priority encoder and upshifter. Therefore the signal did not have enough of an oscillation to be heard when the divider was not a power of two.

Additionally, there was a significant amount of noise that was introduced to the signal. This was proportional to the amplitude of the signal. Most of the noise was filtered out using an analog low pass filter circuit, but a significant amount still remained. Future work looking into solutions for this would include the implementation of a better filter as well as looking into the effects of DAC latching on the signal output. The drops and increases in the signal noise occurred at the same time the ready signal changed. Becuase the DAC controls were tied to ready through the FPGA, we feel that the latching could possibly be a large reason for the introduced noise, but the data sheet for the AD558 did not talk much about the electrical limitations of the chip. Unfortunately, this means the signal still does not sound like a completely pure tone, but the note is definitely recognizable and had an interesting computerized quality.

References

[1] http://www.midi.org/techspecs/index.php
[2] http://www.ibiblio.org/emusic-l/info-docs-FAQs/MIDI-doc/MIDI-Statusbytes.txt
[3] http://www.phys.unsw.edu.au/jw/notes.html
[4] http://www.xilinx.com/ipcenter/index.htm

Parts List

Description	Part Number	Manufacturer	Supplier	Quantity	Price (USD)
MIDI Cable	DIN 5 M/M		CableWholesale	1	1.75
Keyboard	O2 MIDI USB	M-Audio	M-Audio	1	100
Optocoupler	6N138QT-ND	Fairchild Optoelectronics Group	Digikey	1	1.16
DAC	AD558KNZ-ND	DACPORT	Digikey	1	18.02
Opamp	LM386	National Semiconductor	Digikey	1	1.01
220Ω Resistor	690700	CIC Components	Jameco	1	0.015
270Ω Resistor	690726	CIC Components	Jameco	1	0.015
1.2kΩ Resistor	690881	CIC Components	Jameco	1	0.015
1kΩ Resistor	690865	CIC Components	Jameco	1	0.015
390Ω Resistor	690769	CIC Components	Jameco	1	0.015
0.1uF Capacitor	25523	Sunrom Technologies	Jameco	3	0.08
0.047uF Capacitor	57621	Panasonic	Digikey	1	0.12
10 uF Capacitor	10882	Panasonic	Jameco	1	0.015
Speaker	57RF05	Std Intl HK Limited	Load Parts	1	10

Appendix A: Overview of the MIDI Protocol

The Musical Instrument Digital Interface (MIDI) protocol, established in 1983, is a serial transmission standard for control signals relevant to musical instruments. It transmits little-endian 8-bit words plus one start bit and one stop bit at 31250 baud over DIN-5 cables with 20mA current at 5.4VDC, idling at logic high. A complete MIDI message is three bytes. The first byte, known as the status byte, contains the type of message and the channel number. The remaining bytes contain values pertinent to the message type. For note-on messages, the second byte contains the note number, and the third byte contains note velocity. Control messages, triggered by the knobs and faders on a control surface, contain the control number in the second byte and the control value in the third. A note-on with zero velocity is equivalent to a note-off.

MIDI was designed specifically for audio control applications, and it is reflected in the standards it calls for. The protocol specifies optocoupler circuits on every receiver, to prevent ground loops from forming with potential to cause audible distortion, and to separate transmission voltage and current from the processing logic. As indicated by the high baud rate, the MIDI protocol was designed so that a sequence of messages, for example the keys making up a chord on a keyboard, could appear to be sent simultaneously. To this end, the protocol supports “running status” messages, where one status byte gives context for a series of data byte pairs. For example, when a musician plays a chord, the keyboard could eliminate overhead by transmitting a single status byte, indicating note-on and its channel, followed by only the note number and velocity bytes for each key pressed. Since all status bytes begin with a 1, and thus are greater than or equal to 0x80, while data values are limited to 128 values from 0x00 to 0x7F, running status support can be implemented without excessive difficulty.

For more information about MIDI, please refer to the complete MIDI specification listed under References [1].

Appendix B: Generating Phase Step Values

The following code, written in Python, was used to generate the phase step values (dphase) corresponding to each note number.

```
bits = 13      % Bit width of phase value

fs = 44100     % Sample rate (Hz)


for i in range(51,109):              % Valid note numbers for our implementation

       f = 440*pow(2,(i-69)/12.0)    % Compute frequency, with A440 (note 69) as reference

       dphase = int(round(pow(2,bits)*f/fs))      % Compute dphase = f/fs * 2^bits

       print str(dphase)+", "        % Print output in format for data table
```

Appendix C: Verilog for FPGA Implementation

```
/////////////////////////////////////////////////////////////////////////////////
// Leo Altmann and Madeleine Ong
// E155 Final Project
// MIDI Synthesizer
//     Fall 2009
/////////////////////////////////////////////////////////////////////////////////
module top(    input                 ready,
                              input                        noteoo,
                              input                 clk,
                              input    [12:0] theta,
                              input    [5:0]  dialone,
                              output   reg [7:0]   crushed,
                              output   reg [1:0]     adcctrl); // cebar,csbar

       wire [7:0]  sinevalue;
       wire [7:0]  newcrushed;
       wire [1:0]  controls;

       //      singen si(theta,sinevalue);
       singenagain    si(theta, clk, sinevalue);
       // bitcrush (divider)
```

```
	bitcrush	bc(sinevalue, dialone, clk, newcrushed);

	// logic
	assign controls = {noteoo, ready};
	assign adctrl = ready;

	always @(*)
		case (controls)
			2'bx1: begin			// not valid
				adcctrl <= ready;	// latch the value
				end
			2'b00: begin			// note off
				crushed <= 8'b0;	// silence!
				adcctrl <= 2'b0;	// transparent
				end
			2'b10: begin			// note on
				crushed <= newcrushed; // transmit data
				adcctrl <= 2'b0;  // transparent
				end
			default: begin
				crushed <= 8'b0;
				adcctrl <= 2'b0;
				end
		endcase

endmodule


////////////////////////////////////////////////////////////////////////////////
// Leo Altmann and Madeleine Ong
// E155 Final Project
// MIDI Synthesizer
//	Fall 2009
// Module: Bitcrush
// Function: Takes in a sine value, decreases its resolution by dividing it down
// then scaling it back up with the same factor. Also shifts data into the positive
// range (un-two's-complements it)
////////////////////////////////////////////////////////////////////////////////
module bitcrush(	input	[7:0] dividend,
					input  [5:0] divisor,
					input			clk,
					output  [7:0] crushed);

	wire [5:0] remd;
	wire [7:0] quot;
	wire [7:0] shiftone;
	wire [7:0] shifttwo;
	wire [7:0] shiftthree;
	wire [7:0] shiftfour;
```

```
        wire [7:0] shiftfive;
        reg  [7:0] newcrushed;

        // dividor(sinevalue, dialcontrol, dividedvalue, remainder, clk, readyfordata)
        divider di(dividend, divisor, quot, remd, clk, rfd);
        // rescales data (dividedvalue, dialcontrol, clk)
        multiplier mu(quot,divisor, clk);

        assign crushed = newcrushed + 8'b01000000;

endmodule
```

Appendix D: PIC Program Code

```
/*  midisynth.c -- MIDI-Controlled Digital Synthesizer
    Leo Altmann <laltmann@hmc.edu>
    Madeleine Ong <mong@hmc.edu>
    ENG 155 Final Project
    Created:    11/19/2009
    Modified:   12/08/2009

    Receives MIDI messages, taken from an optocoupler output.  Decodes note on messages that
    determine pitch, and control messages to change the bitcrusher parameters.  This program
    sends 22 bits to the FPGA: 13-bit phase value for sinusoid generator, 7-bit divisor for
    bitcrusher routine, note on and data ready.


    DIE CODES:
    0xFA:  Bad interrupt

*/

#include <p18f4520.h>
#include <stdio.h>
#include <stdlib.h>

// Prototypes
char getCharMidi(void);
void isr(void);
void processMessage(void);
void die(char message);
rom unsigned int phasesteps[];

// Globals
#define SAMPLETIME  143     // Experimentally determined for 44.1kHz, timer0 w/ no prescalar
#define STATUSMASK  0xF0    // For masking channel voice message
#define ONMASK      0x01   // Sets note on signal
```

```
#define OFFMASK     0xFE   // Clears note on signal
#define READYMASK   0x02 // Data ready flag on
#define UNREADYMASK 0xFD    // Data ready flag off
#define DIEMASK     0x0C   // Exception alert
#define CRUSHCONTROL 73     // Which rotary controls the bitcrusher

#define NOTE_ON     0b1001  // Op-codes from status bytes
#define NOTE_OFF    0b1000
#define CONTROL     0b1011

unsigned int phase = 0x0000;    // Actually only using 13 bits
unsigned int dphase = 0x009A;   // Phase step
unsigned char crush = 1;        // Divisor for bitcrusher

unsigned char statusByte = 0;
unsigned char dataByte1 = 0;
unsigned char dataByte2 = 0;

unsigned char midiIn = 0;       // New midi byte
unsigned char currNote = 0;
unsigned char lastNote = 0;
char rcvState = 0;              // MIDI receiver state
char lastOp = 0;                // Most recent status byte

// Interrupt vector
#pragma code high_vector = 0x08

void high_interrupt(void) {
    _asm
        GOTO isr
    _endasm
}

// Main Functions
#pragma code

void main(void) {

    // PIC Configuration

    /*
    PORTA
    5  unused
    4:0  phase out MSBs

    PORTB
    5:2  DEBUG LEDs
    1  DATA READY
    0  NOTE ON
```

```
PORTC
7  MIDI IN (USART)
6:0  crush out

PORTD
7:0  phase out LSBs

PORTE
unused

PORTC, PORTD and PORTE map directly from the PIC to the FPGA.

*/
TRISA = 0b00000000; // Used for debug LEDs
TRISB = 0b00000000;
TRISC = 0b10000000; // Use PORTC[7] for USART input
TRISD = 0b00000000;
TRISE = 0b00000000;

RCSTA = 0b10010000; // Enable USART, in 8-bit mode, x, enable receiver,no address detect,
                    // clear framing error, clear overrun error, clear 9th bit.
TXSTA = 0b01000100; // x, 8-bit mode, disable transmit, asynch mode,??, high speed,x, x
BAUDCON = 0b00001000; // clear,clear 0,x,16-bit mode,0, sample continuously, no auto baud

SPBRG = 159;       // 31250 baud, 16-bit asynch., high speed
                    // each byte is 320 us at 31250 baud

                    // Timer0: want 113.38 instructions per sample
T0CON = 0b11001000; // Enable timer, 8-bit mode, internal clock, low-hi transition,
                    //    no prescalar
TMR0L = SAMPLETIME; // Load and start the timer

INTCON = 0b10100000; // Use interrupts, enable timer 0 interrupt
WDTCON = 0;            // Disable watchdog timer

PORTA = 0;
PORTD = 0;
PORTB = 0x03;      // Debug LEDs

while (1) {

    // MIDI Decoder FSM
    // Re-written for run-time efficiency and robustness.
    switch (rcvState) {

        case 0:  // First byte of message
            midiIn = getCharMidi();
            //PORTD = 0xF0;
```

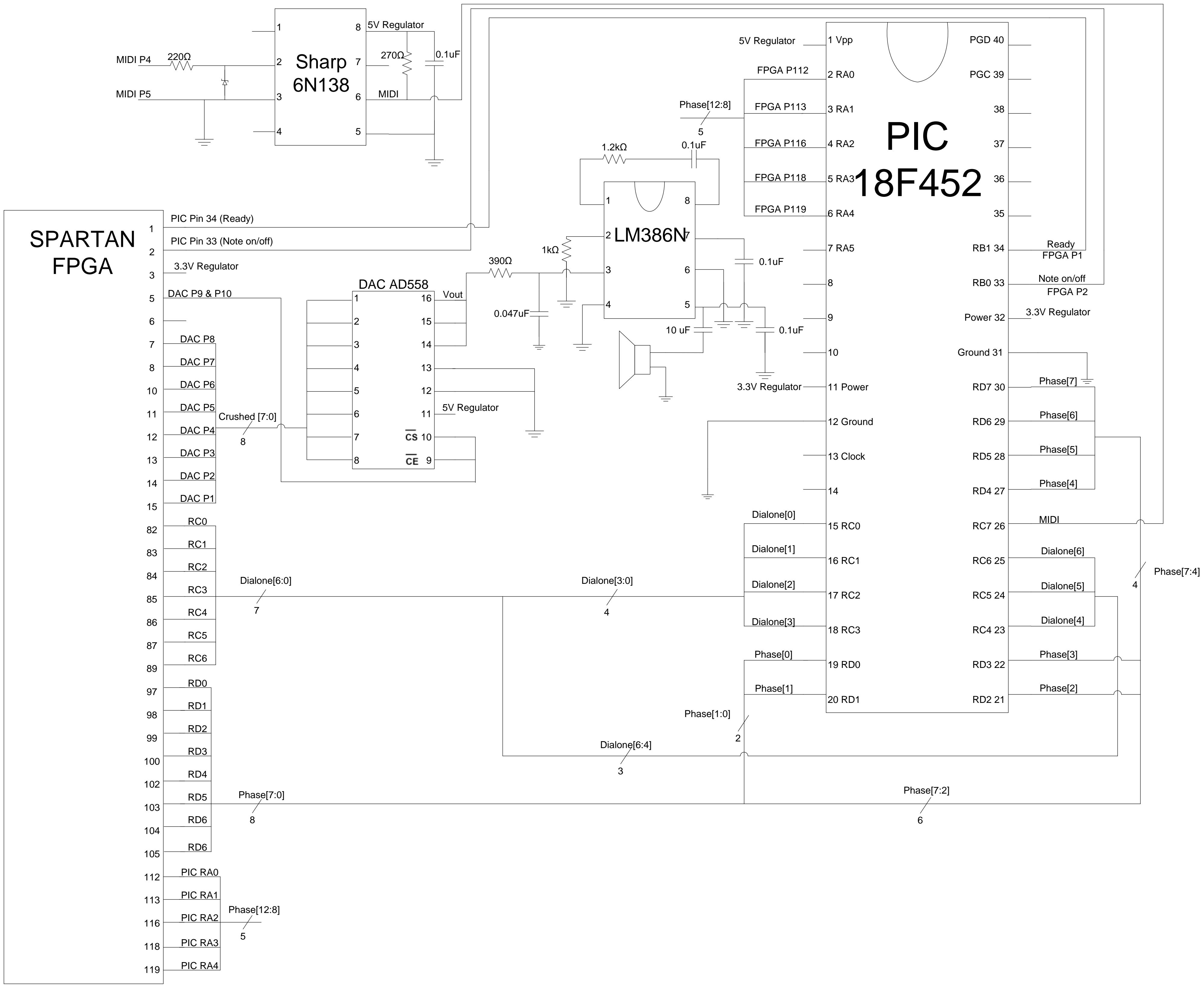
```
                if (midiIn > 127) {
                    statusByte = midiIn >> 4;
                    rcvState = 1;
                    PORTA = 1;
                    break;
                }   // else: message began with data byte, roll into next state

            case 1:  //  First data byte of message
                // Only get new byte if old one is processed
                if (rcvState == 1) { midiIn = getCharMidi(); }

                dataByte1 = midiIn;
                rcvState = 2;
                break;

            case 2: //  Second data byte of message
                midiIn = getCharMidi();
                dataByte2 = midiIn;
                processMessage();   // Decode message

            default: rcvState = 0;
        }
    }
}

void processMessage(void) {
    // Decodes complete MIDI messages after reception
    switch (statusByte) {
        case NOTE_ON:
            if (dataByte2 > 0) { // Zero velocity denotes note off
                lastNote = currNote;
                currNote = dataByte1;
                dphase = phasesteps[currNote];
                PORTB = PORTB | ONMASK;
                break;
            }   // If it was really note off, roll into next case

        case NOTE_OFF:
            if (lastNote) {
                if (lastNote == dataByte1) {
                    lastNote = 0;
                } else {
                    currNote = lastNote;
                    lastNote = 0;
                    dphase = phasesteps[currNote];
                }
            } else {
                currNote = 0;
                PORTB = PORTB & OFFMASK;
```

```
            } break;

        case CONTROL:
            if ((dataByte1 == CRUSHCONTROL) & (dataByte2 > 0)) {
                crush = dataByte2;
            } break;

        default: break;   // Discard byte
    }
}

char getCharMidi(void) {
    // Poll the UART for a new MIDI character input

    while (1) {        // Extra loop to ensure heartbeat bytes don't disrupt the system
        while (~PIR1bits.RCIF) {} // Poll for new byte

        if (RCSTAbits.FERR)    {   // Framing error
            PORTB = PORTB | DIEMASK;
        }
        if (RCREG != 0xFE) {  // Ignore bad bytes
            return RCREG;
        }
    }
}

void die(char message) {
    // Some fatal error has occured.
    // Stop the program and give the programmer some feedback!
    INTCON = 0;             // Disable interrupts
    TOCON = 0;              // Disable timer0
    PORTD = message;    // Alert programmer
    PORTB = PORTB | DIEMASK;
    while (1) {}       // Do no more harm
}

#pragma interrupt isr

void isr(void) {

    if (INTCONbits.TMR0IF) {   // Timer 0 interrupt
        PORTB = PORTB & UNREADYMASK;    // Clear ready flag
        INTCONbits.TMR0IF = 0;            // Clear interrupt flag
        PORTD = phase;                  // Send data to FPGA
        PORTA = (phase >> 8);
        PORTC = (crush & 0x7F);
        phase += dphase;                // Update the wave phase
        TMR0L = SAMPLETIME;             // Restart sample timer
        PORTB = PORTB | READYMASK;      // Set ready flag
```

```
    } else {
        // How did we get here?
        die(0xFA);
    }
}

#pragma code

rom unsigned int phasesteps[] = {
    // Table of values for dphase, indexed by note number
    // Zeros for unused note numbers prevent address errors

    // A4 (440) = MIDI note 69
    // Frequency = 440 * pow(2.0,(note-69.0)/12.0);
    // dphase = pow(2, BITS) * frequency / FS;
    // BITS = 13, FS = 44100

    0, 0, 0, 0, 0, // 0-4
    0, 0, 0, 0, 0, // 5-9
    0, 0, 0, 0, 0, // 10-14
    0, 0, 0, 0, 0, // 15-19
    0, 0, 0, 0, 0, // 20-24
    0, 0, 0, 0, 0, // 25-29
    0, 0, 0, 0, 0, // 30-34
    0, 0, 0, 0, 0, // 35-39
    0, 0, 0, 0, 0, // 40-44
    0, 0, 0, 0, 0, // 45-49
    0, 29,  31,  32,  34, // 50-54
    36,  39,  41,  43,  46, // 55-59
    49,  51,  55,  58,  61, // 60-64
    65,  69,  73,  77,  82, // 65-69
    87,  92,  97,  103,  109, // 70-74
    116,  122,  130,  137,  146, // 75-79
    154,  163,  173,  183,  194, // 80-84
    206,  218,  231,  245,  259, // 85-89
    275,  291,  309,  327,  346, // 90-94
    367,  389,  412,  436,  462, // 95-99
    490,  519,  550,  583,  617, // 100-104
    654,  693,  734,  778       // 100-104

};
```

Appendix E: Breadboarded Schematic

The circuit was constructed in six modules: optocoupler circuit, PIC, FPGA, DAC and amplifier circuit for the speaker.



Sharp
6N138
MIDI P4
MIDI P5
220Ω
5V Regulator
270Ω
0.1uF
MIDI
PIC
18F452
LM386N
1.2kΩ
0.1uF
1kΩ
390Ω
0.047uF
10 uF
0.1uF
DAC AD558
Vout
CS
CE
SPARTAN
FPGA
PIC Pin 34 (Ready)
PIC Pin 33 (Note on/off)
3.3V Regulator
DAC P9 & P10
DAC P8
DAC P7
DAC P6
DAC P5
DAC P4
DAC P3
DAC P2
DAC P1
Crushed [7:0]
RC0
RC1
RC2
RC3
RC4
RC5
RC6
RD0
RD1
RD2
RD3
RD4
RD5
RD6
RD6
PIC RA0
PIC RA1
PIC RA2
PIC RA3
PIC RA4
Phase[12:8]
Dialone[6:0]
Dialone[3:0]
Dialone[6:4]
Phase[7:0]
Phase[7:2]
Phase[1:0]
Phase[7:4]
1 Vpp
2 RA0
3 RA1
4 RA2
5 RA3
6 RA4
7 RA5
11 Power
12 Ground
13 Clock
15 RC0
16 RC1
17 RC2
18 RC3
19 RD0
20 RD1
PGD 40
PGC 39
RB1 34
RB0 33
Power 32
Ground 31
RD7 30
RD6 29
RD5 28
RD4 27
RC7 26
RC6 25
RC5 24
RC4 23
RD3 22
RD2 21
FPGA P112
FPGA P113
FPGA P116
FPGA P118
FPGA P119
Ready
FPGA P1
Note on/off
FPGA P2
Dialone[0]
Dialone[1]
Dialone[2]
Dialone[3]
Dialone[4]
Dialone[5]
Dialone[6]
Phase[0]
Phase[1]
Phase[2]
Phase[3]
Phase[4]
Phase[5]
Phase[6]
Phase[7]