# MIDI Sound Module

Final Project Report
December 11, 2009
E155

Aaron Guillen and Allison Russell

**Abstract:**

A MIDI sound module was implemented to interpret MIDI messages from a keyboard and then output the sampled waveform of another instrument. An FPGA was configured as a UART to receive MIDI messages from a keyboard. A PIC was programmed to read in and interpret MIDI messages to send note values and to receive the samples of an instrument's waveform from a PC. The waveform samples were converted to an analog signal and conditioned for a commercial audio amplifier. The module successfully outputs the waveforms at 3kHz, producing recognizable but rather low quality notes of a bass guitar.

**Introduction and Overview:**

MIDI is a music communication protocol in which music information is stored in bytes. The bytes hold op-codes that indicate basic instructions such as note on or off, note value and volume, as well as various special effects. Any MIDI device can be configured to output these bytes through a MIDI cable to be interpreted by another device. The purpose of this MIDI module is to handle the note on, note off and note value information to output a waveform of the corresponding note of another instrument. The module uses a keyboard as the MIDI controller. The keyboard sends three bytes every time a note is pressed or released, which are sent to a UART implemented on the FPGA. The FPGA separates the three bytes and sends them to the PIC for interpretation. The PIC ignores any message that is not note on or note off. The PIC is configured to asynchronously transmit and receive messages with a PC using a USB serial converter. When a note should be off, the PIC sends zeros to the PC. When a note should be on, the PIC sends the correct note value to the PC. The PC sends back three samples of an instrument's waveform at 1kHz. The PIC receives them in memory and outputs samples to the FPGA at 3kHz, the original rate the instrument's waveform was sampled at. The samples are output a nybble at a time. The FPGA deserializes the nybbles and outputs a sample byte to a digital to analog converter. The analog signal is conditioned with an op-amp and is sent to an amplifier. The result is a keyboard that can produce notes from another instrument. A block diagram representation of the project can be found in Appendix A. This report will cover the details of the design.

**New Hardware:**

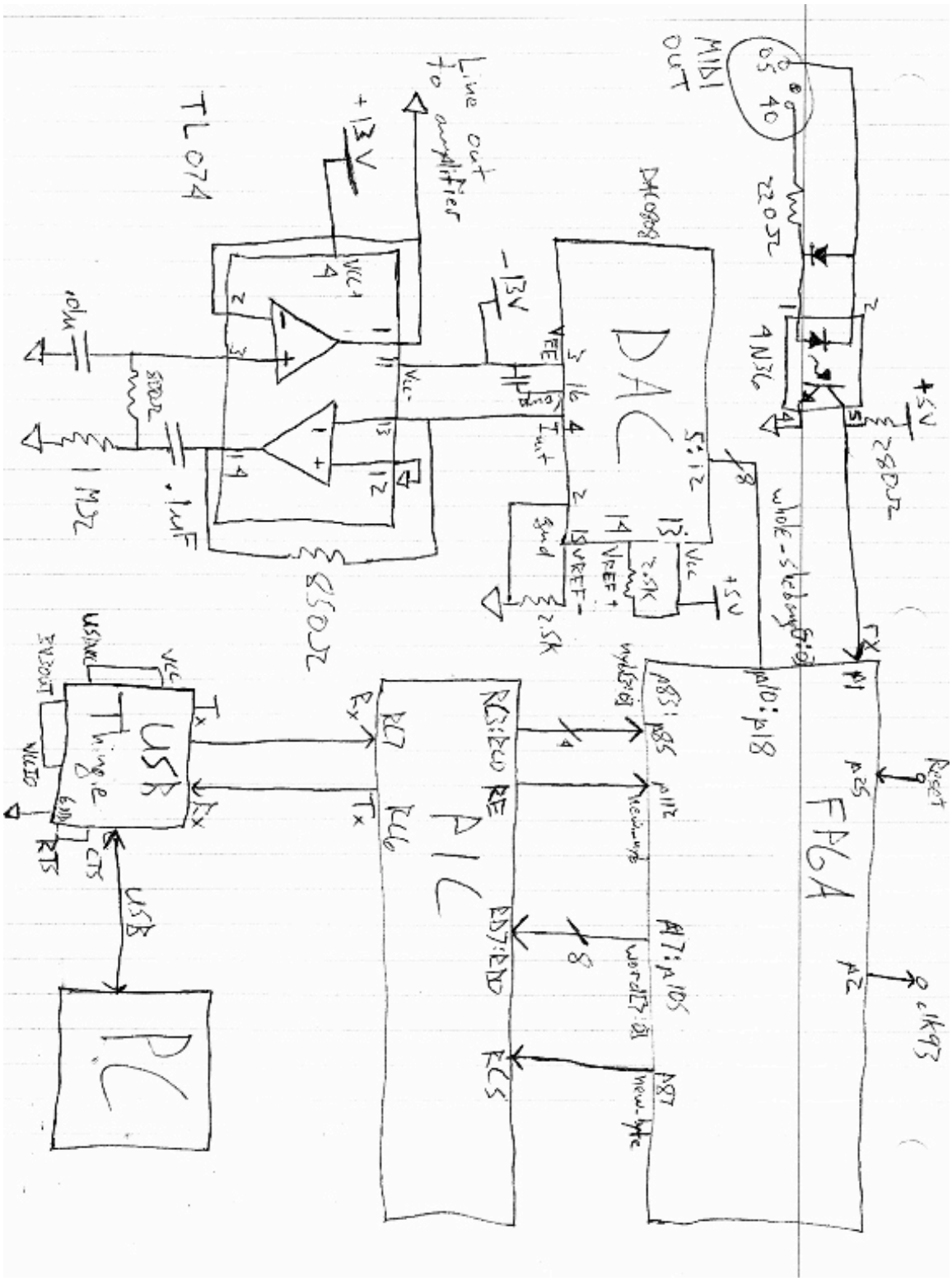
The project used three pieces of new hardware in addition to Xilinx Spartan FPGA and PIC 18F4520, described below.

USB Thingie: The MIDI module uses a USB Thingie manufactured by Smiley Micros to interface with a PC. The USB Thingie connects to a USB port on a PC and connects to some peripheral device, the PIC in this case, through a UART. It uses a common USB to UART IC made by Future Technology Devices International. The driver that is provided by FTDI allows the USB Thingie to show up as a Virtual Com Port in Windows. After installation of these drivers, the USB Thingie can be controlled in exactly the same manner as the BlueSMiRF module used in earlier e155 labs. The advantage of the USB Thingie is that it can operate at much higher baud rates than Sparkfun's bluetooth Dongle.

MIDI and the 4N36 optocoupler: MIDI devices output messages as currents, which must be electrically isolated before they are sent to the Harris Board. Several subtle problems may occur when setting up the electrical isolation for the MIDI serial connection. When connecting an optocoupler with an internal LED and phototransistor, the digital power supply should be connected to the collector of the transistor through a pull-up resistor (Pin 5 of the 4N36 made by Fairchild Semiconductor). The emitter (Pin 4) should be connected to ground. The most important part is to leave the base floating (Pin 3). Light from the LED that hits the base will inject current into the base to turn on the transistor. Pin 5 of the MIDI cable should be connected to the LED's cathode (Pin 2) and pin 4 should be connected to the anode (Pin 1) through a current limiting resistor. This circuit conforms to the electrical specifications provided by the MIDI Manufacturer's Association.

DAC0808: The DAC 0808 is a current output 8-bit DAC with an error of 1LSB. It requires a +5V (Pin 13) and -10V to -15V power supply (Pin 3). A resistor should be placed in between the reference voltage power supply and the positive reference voltage pin (Pin 14). The value of this resistor should be chosen so that when the pin is pulled to ground during maximum output (i.e., when the input is 0xFF), the current into the pin is 2mA. The same resistance should be placed between ground (Pin 2) and the negative reference voltage (Pin 15). A bypass capacitor should be placed between the negative power supply and the compensation input (Pin 16). The DAC outputs a current between 0 and 2mA.

**Schematic:**


MIDI OUT
DAC
TL074
Line out to amplifier
+13V
-13V
FPGA
PIC
USB
PC
Reset
Clock

**Reading the MIDI messages on the FPGA:**

The first step in communicating with a MIDI controller is to electrically isolate the MIDI cable from the receiving device using a phototransistor optocoupler. Current sent over the MIDI cable turns on an LED inside the optocoupler. The light from this LED causes current to flow into the base of the phototransistor. The transistor is turned on and the output is pulled low. This output is sent to a UART to be deserialized.

Because the PIC's UART is being used for communications with the PC, the serial voltage signal from the optocoupler circuit is sent to an I/O pin on the FPGA, which has been programmed to act as a UART. The UART reads the MIDI message, de-serializes it and sends it a byte at a time to the PIC. The UART takes in the Harris Board's clock, a universal reset and the MIDI voltage signal, and outputs a byte of data. Verliog code for the UART is in Appendix B. The UART is constructed of the four sub-modules described below.

The clock divider module, divclk, creates a 93kHz clock. 93kHz was chosen because it is approximately three times the baud rate for MIDI. The rest of the UART uses this clock to sample each bit of the MIDI signal three times. The majority of these samples is taken to reconstruct the signal.

The synchronizer module, sync, takes in the asynchronous MIDI voltage input and uses two registers to synchronize the input with the 93kHz clock of the UART. The synchronizer is necessary to avoid metastable bits.

The oversample module, oversample, clocked by the 93kHz clock, shifts in three bits of the input signal. The majority of these three bits is found using xnor gates and is output to another module as one bit of the MIDI byte. Because the 93kHz clock is close to, but not exactly, three times the baud rate of the MIDI signal, the oversampler sometimes becomes misaligned. Ideally, the module would always take three samples of the same bit of the input signal, however when it is misaligned it occasionally takes two samples of a first bit and one of a second, or one sample of a first bit and two of a second. In the first case, the module should sample just two times in the next cycle to become realigned with the MIDI clock. In the second case, the module should count four samples before taking the majority so it can move out the odd bit and take three samples from the same bit. A submodule called urescreator handles the counter, which controls when the oversampler stores the majority of the bits in the three registers. This counter is adjusted by a mux which is conditioned by which bits were equal in the previous majority detect. If they were all the same, it counts to three on the next cycle. However if the first was different or the third was different, it counts to four or two respectively. When the counter overflows, urescreator asserts an output signal "ures" which indicates when a new bit has been created. "Ures" enables a shift register in the oversample module to register the majority. The registered bits are sent to an eight bit shift register to be deserialized.

The shift register module, shiftreg, takes in the bits serially from the averaged samples of the oversampler. There are eight registers so that the shift registers will eventually hold a complete byte of MIDI data. The shift register is enabled by the "ures" signal so the bits shift on the clock edge, only when there is a new bit. The registers constantly shift but are not loaded into the output byte's buffer until a signal "send" is asserted. "Send" is an output of a sub-module finite state machine (FSM) called

serialcontrol. The FSM in serialcontrol stays in an idle state until it receives the start bit. Since MIDI idles at logic level one, the FSM waits for logic level zero as a start bit. When the state machine receives the start bit, it moves through states every time a new bit is output by the oversampler. When all 8 data bits have been loaded, the FSM asserts "send" which tells shiftreg to load the contents of the shift registers into an 8 bit output register. This bus is the output of the UART, which is then sent to the PIC.

When the UART on the FPGA has extracted a byte of MIDI data, it writes it to PORTD on the PIC. Simultaneously, the FPGA toggles pin 5 on PORTC to indicate that a new byte is available. The PIC polls this pin and stores the contents of PORTD whenever it changes.

**Interpreting messages on the PIC:**

The MIDI messages have three bytes: status, note and volume. The PIC is programmed to interpret these messages. C code for the PIC can be found in Appendix C. When the PIC first receives a byte, it checks the most significant bit, since only the status messages will have a one in the most significant bit. If the most significant bit is one, the value of PORTD is stored in a global variable "status" and a counter is set to zero. Before pin 5 is polled again, the PIC will check the value of status. The only messages of interest are note on "0x90", and note off "0x80". If status is note on, the PIC will set a variable "valid" to 1. If status is note off, "valid" will be set to 2. Otherwise, "valid" will be zero.

The next time the FPGA indicates a new byte by toggling pin 5, the PIC will check the values in counter and valid. If counter is zero, the status byte was just loaded so the current byte is note. When valid is one, this note should be on so the PIC sends the byte in PORTD to TXREG to be transmitted to the PC. When valid is two, this note should be off so the PIC sends a null value to TXREG. In both cases, the counter is incremented so that the volume byte of the MIDI message is skipped and the PIC is ready to receive the next three bytes.

**PIC and LabView Communications:**

The PIC is connected to the PC through a USB Thingie serial converter. When the PIC loads a note byte into TXREG, the built in USART transmits it to the receive port on the USB Thingie. A LabView VI is configured to read in the note value and send back samples of a waveform to the PIC. The block diagram of the LabView VI can be found in Appendix D. The LabView VI reads the COM port of the USB Thingie with a built in block called VISA read, and stores the note byte in the read buffer.

The content of the read buffer is stored as the current note and a counter is initialized to zero. Each iteration of the main while loop requires 1ms to execute, so to ultimately output samples at 3kHz, the loop must send three samples over the UART with each cycle. The sampled waveforms are read from a tab-delimited file at the startup of the LabView VI and are stored in an array in the computer's memory. The current note value is used to index the array's columns, and the counter is used to index the rows. The

current note value is scaled so that the lowest note message will correspond with column zero. Since three samples are sent on each loop iteration, LabView indexes three consecutive samples (rows) and sequentially writes them to the serial port's write buffer. The counter is then incremented by three. On the next iteration, LabView looks to see if a new note was received. If not, it continues outputting the current note's waveform. If the current note is either zero (before being scaled) or not contained in the array, LabView simply outputs zeros for each of the three samples.

The PIC receives these samples one at a time through the USART. These values are stored in an array in the PIC's memory. A counter, smp_count, is incremented each time a value is received. This counter is used to index which element of the array should be written. When three samples have been counted, the counter is reset.

**Outputting a Waveform:**

The PIC uses timer 0 to control when it outputs the samples. The interrupt of the timer is not enabled, but its flag is polled so that the PIC can send a sample whenever the timer overflows. Every time the flag goes high, the PIC reinitializes the timer and retrieves the sample at the memory address of the array pointer plus an index counter. The index counter is incremented whenever the PIC outputs a byte and is reset whenever the third sample is output. To remove the discontinuity from being stored as two's complement bytes, 0x80 is added to each sample. The normalized sample is then output to the FPGA.

The PIC has two 8 bit ports, PORTC and PORTD. Since all of PORTD and half of PORTC are occupied, the sample byte must be output in two nybbles to the FPGA. The lower four bits of PORTC are configured as outputs. The PIC sends the lower nybble of the sample to PORTC and toggles an indicator bit on PORTE. The sample is then downshifted so the upper nybble is in the lower four bits, and is sent to PORTC. The indicator bit is toggled again. The FPGA is programmed to deserialize these nybbles into one byte using a shift register that is enabled by output "deser" of a finite state machine in module desernybs. The FSM sits in an idle state until it the bit from the PIC toggles. It then moves to a state indicating that it has received the first nybble. It stays in a holding state until the PIC bit toggles again, indicating it has received the second nybble. Signal "deser" is set high when the FPGA has either the first or the second nybble. When both nybbles have been received, signal "whole" is set high. This enables the FPGA to register the contents of the shift registers into one concatenated byte. This byte is connected to a DAC.

The DAC converts the FPGA's output byte into an analog current output. This current ranges from 0 to 2mA and needs to be conditioned before being sent to an audio amplifier. The circuit shown in the schematic sends this current across an 850Ω resistor to convert it into a voltage at the output. 850Ω is chosen so that the peak-to-peak voltage of this signal is 1.7V, which is the standard line level for many American audio devices.

The output of this circuit is fed into a passive RC bandpass filter. The filter removes both low and high inaudible frequencies from the signal. Removal of the DC level and low frequency components prevents constant deflections of the speaker cone, which may distort the audio. The components were chosen so that the corner frequencies

of the band-pass filter are 15 Hz and 20 kHz. A voltage follower buffers the filter's output to provide a low-impedance signal to an audio amplifier. Texas Instruments' TL074 quad operational amplifier is used for these circuits.

**MatLab and Data Storage:**

Three octaves of bass guitar were recorded in thirty-seven sound files. They were all recorded and saved in a linear pulse code modulated .wav file to avoid logarithmic scaling and other lossy compression methods. This file format stores each 16-bit sample of the waveform in an array. A MatLab routine returns the list of samples in the file. The original files were recorded at 44.1 kHz. To produce an array of 3 kHz samples, every fifteen samples were averaged. The introduced a sampling rate error which pitch shifted all of the notes by a quarter-tone.

Next, all of these averaged samples were normalized to 8-bit values. The maximum absolute value of the samples in each array was determined and the entire array was scaled so that the maximum value was 127. These samples were then rounded to the nearest integer between -128 and 127. This was performed for every .wav file, and these lists of samples were stored as the columns of a tab-delimited text file.

**Results:**

The MIDI sound module successfully outputs the waveforms of a bass guitar. Unfortunately, the PIC could not run fast enough to send the samples at 44.1kHz, the original sample rate of the notes. Therefore, the sample output rate had to be lowered to 3kHz. This caused distortion of high frequency signals.

In retrospect, this problem could have been avoided by implementing everything on the FPGA. The FPGA was already programmed with a UART. If it was programmed with a second UART, it could have read in 44 samples from LabView and stored them in its own RAM. The PIC was only fast enough to read in 3 samples. Since the FPGA has a divider and many multipliers, the signal could have been multiplied by the MIDI volume message and windowed with the divider to produce a less distorted sound. All of this could have occurred simultaneously on the FPGA, while the PIC was constrained to one operation per instruction cycle. Aside from running slowly, the system worked as intended.

**References:**

[1] “Tech Specs and Info,” MIDI Manufacturer’s Association. http://www.midi.org/techspecs/index.php

[2] “DAC0808,” National Semiconductor. http://www.national.com/mpf/DA/DAC0808.html#Overview

[3] “4N36,” Fairchild Semiconductor. http://www.fairchildsemi.com/pf/4N/4N36-M.html

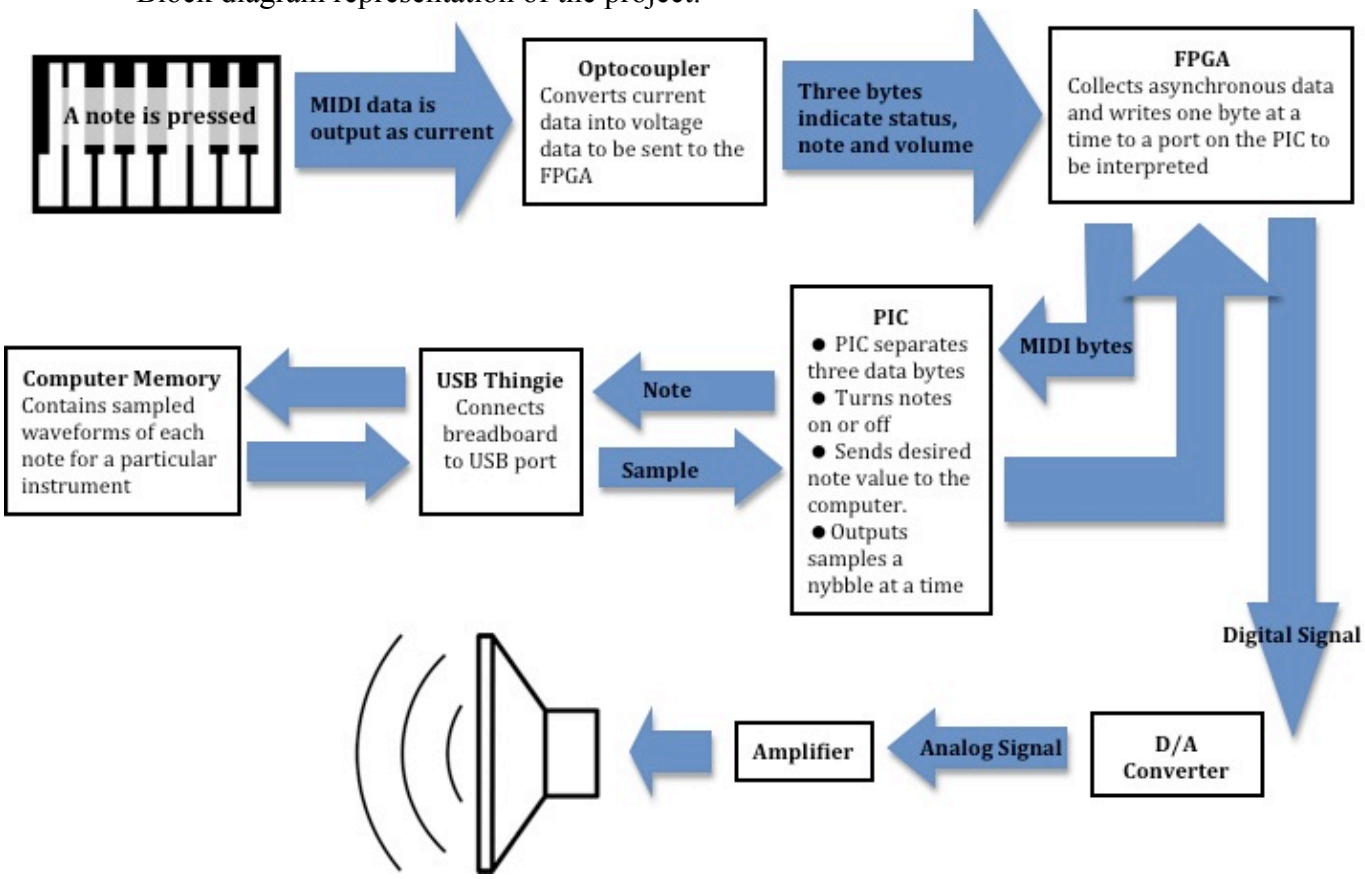
[4] “USB meets Breadboard,” Smiley Micros. http://www.smileymicros.com/index.php?module=pagemaster&PAGE_user_op=view_page&PAGE_id=31

[5] National Instruments Help Forums. http://forums.ni.com/ni/board/message?board.id=170&thread.id=282110&view=by_date_ascending&page=1

**Parts List:**

| Part | Source | Part Number | Price |
|---|---|---|---|
| USB Thingie | Smiley Micros | N/A | $26.90 |
| Optocoupler | West Florida Components | 4N36 | $0.30 |
| MIDI Cable | Style’s Music 777 E Foothill Blvd, Pomona | N/A | $8.73 |
| DAC | Stock Room | DAC0808 | $0 |
| Op-Amps | Stock Room | TL074 | $0 |
| Audio Amplifier | Personal | N/A | $0 |

**Appendix A:**
Block diagram representation of the project.



**Appendix B:**
Verilog Code

```
//////////////////////////////////////////////////////////////////////////////////
// Company: Harvey Mudd College
// Engineer: Aaron Guillen and Allison Russell
//
// Create Date:            19:44:36 11/15/2009
// Design Name:            UART and Deserializer
// Module Name:            mpproj_aarg
// Project Name:           MIDI Sound Module
// Description:    Top level module for a UART on the FPGA.  Also receives
// samples as nybbles from the PIC and deserializes them to output waveform
//
//////////////////////////////////////////////////////////////////////////////////
module mpproj_aarg(
        input clk,
        input rx,
        input reset,
        input [3:0] nyb,
        input newnyb,
        output [7:0] word,
        output clk93,
        output new_byte,
        output [7:0] whole_shebang
   );

        wire clk93;
        wire rxsync;
        wire serbit;
        wire ures;
        // Divide the clk to get a 93kHz clock, 3 times the baud rate for MIDI
        divclk dv(clk, reset, clk93);

        // Because the input is asynchronous, synchronize it to avoid metastability
        sync sn(clk93, reset, rx, rxsync);

        // Oversample the serial input and output the average of 3 data points
        oversample os(clk93, reset, rxsync, serbit, ures);

        // De-serialize the data. Output to word byte to word, sent to PIC
        shiftreg sr(clk93, ures, serbit, reset, word);

        // De-serialize the nybbles.  Output whole shebang to a DAC
        desernybs dn(clk, reset, nyb, newnyb, whole_shebang);
endmodule
```

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer: Aaron Guillen and Allison Russell
// Create Date:    21:09:17 11/15/2009
// Module Name:          divclk
// Description:          divclk takes in the pic's 20MHz clock and divides it with a
// counter so that it is a 93kHz clock.  93kHz was chose because it is three times
// the speed of the MIDI signal clock.
//////////////////////////////////////////////////////////////////////////////////
module divclk(
        input clk,
        input reset,
        output reg clk93
        );

        reg [6:0] counter;
        wire edge93;

        // edge93 is asserted when counter=107
        assign edge93 = counter[6] & (counter[5]) & (~counter[4]) & (counter[3]) &
                (~counter[2]) & (counter[1]) & counter[0];

        // counter resets on "reset" and when it has counted to 107 (edge93=1)
        always @(posedge clk, posedge reset)
                if (reset)
                        counter <= 0;
                else if (edge93)
                        counter <= 0;
                else
                        counter <= counter + 1;

        // clk93 toggles whenever edge93 rises
        always @(posedge edge93, posedge reset)
                if (reset)
                        clk93 <= 0;
                else
                        clk93 <= ~clk93;

endmodule
```

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer: Aaron Guillen and Allison Russell
// Create Date:    20:27:55 11/15/2009
// Module Name:    sync
// Description:    Sync is a synchronizer module that takes in the
// asynchronous MIDI voltage input and uses two registers to synchronize
// the input with the UART 93kHz clock
//////////////////////////////////////////////////////////////////////////////////

module sync(
	input clk,
	input reset,
	input rx,
	output reg rxsync
	);

	reg rx2;

	always @(posedge clk, posedge reset)
		if (reset) begin
			rx2 <= 1;
			rxsync <= 1;
		end
		else begin
			rx2 <= rx;
			rxsync <= rx2;
		end

endmodule
```

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer: Aaron Guillen and Allison Russell
// Create Date:    21:12:26 11/15/2009
// Module Name:          oversample
// Description:          Oversample oversamples the synchronized input and
// outputs the average of the 3 oversampled bits.
//////////////////////////////////////////////////////////////////////////////////
module oversample(
        input clk93,
        input reset,
        input rxsync,
        output reg serbit,
        output ures
   );

        //Declare wires and registers
        wire eq23, eq34, r4p;
        reg r2, r3, r4;
        reg eq23p, eq34p;

        //Call submodule to create ures, a counter which tells the
        //oversampler when to average
        urescreator uc(clk93, reset, eq23p, eq34p, ures);

        //Shift registers for taking in three bits at a time
        //Reset is 1 because MIDI idles at 1
        always @(posedge clk93, posedge reset)
                if (reset) begin
                        r2 <= 1;
                        r3 <= 1;
                        r4 <= 1;
                end
                else begin
                        r2 <= rxsync;
                        r3 <= r2;
                        r4 <= r3;
                end

        //r4p takes the value of r3 instead of r4 on the special
        //case where the first two are not equal but the last two are
        assign r4p = (eq34p & (~eq23p)) ? r3 : r4;
        assign eq23 = r2 ^~ r3;
        assign eq34 = r3 ^~ r4p;

        //Ures is asserted when the oversampler should average the samples
        always @(posedge clk93, posedge reset)
                if (reset) begin
                        eq23p <= 1;
                        eq34p <= 1;
                end
                else if (ures) begin
```

```
                eq23p <= eq23;
                eq34p <= eq34;
        end

    always @(*)
        case({eq23,eq34})
            2'b00:      serbit <= r4;
            default:    serbit <= r3;
        endcase
endmodule

//SUBMODULE
module urescreator(
    input clk93,
    input reset,
    input eq23p,
    input eq34p,
    output reg ures
    );

    reg [1:0] counter;
    wire ur0, ur1, ur2;

    //counter counts up to 4
    always @(posedge clk93, posedge reset)
        if (reset)
            counter <= 0;
        else if (ures)
            counter <= 0;
        else
            counter <= counter + 1;

    assign ur0 = ~counter[0] & counter[1];
    assign ur1 = counter[0] & counter[1];
    assign ur2 = counter[0] & ~counter[1];

    //ures takes on the value of 1, 2 or 3 based on the previous
    //equivalences between bits.
    always @(*)
        case({eq23p,eq34p})
            2'b00:      ures <= ur0;
            2'b01:      ures <= ur2;
            2'b10:      ures <= ur1;
            default:    ures <= ur0;
        endcase

endmodule
```

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer: Aaron Guillen and Allison Russell
// Create Date:    21:26:39 11/15/2009
// Module Name:    shiftreg
// Description:    Shiftreg takes in the serial bits from the oversampler.  when it
// receives a start bit (0), it counts the 8 bits in the byte signal and loads them
// onto an 8 bit output bus to be sent to the PIC.
//////////////////////////////////////////////////////////////////////////////////
module shiftreg(
            input clk93,
            input ures,
            input serbit,
            input reset,
            output reg [7:0] word
    );
            reg intermediate;
            reg [7:0] wordp;
            wire send;
            // FSM shifts bits in every time ures indicates a new bit is in serbit.
            // MSB is sent first over the MIDI serial port
            always @(posedge clk93, posedge reset)
                        if (reset) begin
                                    wordp <= 8'b11111111;
                                    intermediate <= 1;
                                    end
                        else if (ures) begin
                                    intermediate <= serbit;
                                    wordp[0] <= intermediate;
                                    wordp[1] <= wordp[0];
                                    wordp[2] <= wordp[1];
                                    wordp[3] <= wordp[2];
                                    wordp[4] <= wordp[3];
                                    wordp[5] <= wordp[4];
                                    wordp[6] <= wordp[5];
                                    wordp[7] <= wordp[6];
                        end

            //send is a signal from submodule serialcontrol asserted when a byte is complete
                        always @(posedge clk93, posedge reset)
                        if (reset)
                                    word <= 8'b00000000;
                        else if (send) begin
                                    word[7] <= wordp[0];
                                    word[6] <= wordp[1];
                                    word[5] <= wordp[2];
                                    word[4] <= wordp[3];
                                    word[3] <= wordp[4];
                                    word[2] <= wordp[5];
                                    word[1] <= wordp[6];
                                    word[0] <= wordp[7];
                        end
```

```
	serialcontrol sc(clk93, ures, reset, intermediate, send);
endmodule

//SUBMODULE
module serialcontrol(
	input clk93,
	input ures,
	input reset,
	input serbit,
	output send
	);
	//declare parameters for the finite state machine
	reg [4:0] state;
	reg [4:0] nextstate;
	parameter idle = 4'b0000;
	parameter start = 4'b0001;
	parameter MSB = 4'b0010;
	parameter B6 = 4'b0011;
	parameter B5 = 4'b0100;
	parameter B4 = 4'b0101;
	parameter B3 = 4'b0110;
	parameter B2 = 4'b0111;
	parameter B1 = 4'b1000;
	parameter LSB = 4'b1001;

	// FSM loops in idle state until serbit is 0 (start bit =0, stop bit =1)
	always @(*)
		case(state)
			idle:	if(serbit)	nextstate <= idle;
				else			nextstate <= start;
			start:				nextstate <= MSB;
			MSB:				nextstate <= B6;
			B6:				nextstate <= B5;
			B5:				nextstate <= B4;
			B4:				nextstate <= B3;
			B3:				nextstate <= B2;
			B2:				nextstate <= B1;
			B1:				nextstate <= LSB;
			LSB:				nextstate <= idle;
			default:			nextstate <= idle;
		endcase

	always @(posedge clk93, posedge reset)
		if (reset)
			state <= 4'b0000;
		else if (ures)
			state <= nextstate;
	//send is asserted when the state machine is at the least significant bit.
	assign send = (~ures)&(state==LSB);
endmodule
```

```
//////////////////////////////////////////////////////////////////////////////////
// Engineer: Aaron Guillen and Allison Russell
// Create Date:    21:44:30 12/03/2009
// Module Name:    desernybs
// Description:   Desernybs receives nybbles from the PIC and deserializes
// them to output a byte.
//////////////////////////////////////////////////////////////////////////////////
module desernybs(
   input clk,
   input reset,
   input [3:0] nyb,
   input newnyb,
   output reg [7:0] whole_shebang
   );

         //internal signals
         wire deser, whole;
         reg [3:0] highnyb;
         reg [3:0] lownyb;
         reg [2:0] state;
         reg [2:0] nextstate;

         //parameters
         parameter OFF = 3'b000;
         parameter FIRSTnyb = 3'b001;
         parameter CHILL = 3'b010;
         parameter SECONDnyb = 3'b011;
         parameter COMPLETE = 3'b100;

         //deserialize the nybble input from the pic. enable when newnyb toggles
         always @(posedge clk, posedge reset)
                  if (reset) begin
                           highnyb <= 0;
                           lownyb <= 0;
                  end
                  else if (deser) begin
                           highnyb <= nyb;
                           lownyb <= highnyb;
                  end

         //when there is a whole byte, output it
         always @(posedge clk, posedge reset)
                  if (reset)
                           whole_shebang <= 0;
                  else if (whole)
                           whole_shebang <= {highnyb, lownyb};
```

```
	//state machine to create control signals for registers
	always @(posedge clk, posedge reset)
		if (reset)
			state <= OFF;
		else
			state <= nextstate;
	always @(*)
		case(state)
			OFF:	if(newnyb)		nextstate <= FIRSTnyb;
				else			nextstate <= OFF;
			FIRSTnyb:			nextstate <= CHILL;
			CHILL: if(newnyb)		nextstate <= CHILL;
				else			nextstate <= SECONDnyb;
			SECONDnyb:			nextstate <= COMPLETE;
			COMPLETE:			nextstate <= OFF;
			default:			nextstate <= OFF;
		endcase

	//assign control signals
	assign deser = (state == FIRSTnyb) | (state == SECONDnyb);
	assign whole = (state == COMPLETE);

endmodule
```

**Appendix C:**

C code on the PIC.

```
/*
midimodule.c
Created by Aaron_Guillen@hmc.edu and Allison_Russell@hmc.edu, 12/1/09.
Description: This code programs the pic to receive MIDI messages from the
FPGA.  It sends turns a note on or off based on the status message.  It
sends the note that should be played to the PC and receives the samples
back from the PC.  It then outputs the samples a nybble at a time to the
FPGA.
*/

//LIBRARIES
#include <p18f4520.h>
#include <stdio.h>
#include <stdlib.h>
# include <timers.h>
void main(void);
void isr(void);

//INTERRUPT VECTOR
#pragma code high_vector = 0x08
void high_interrupt(void) {
        _asm
                GOTO isr        // Branch to interrupt service routine
        _endasm
}

//INITIALIZE VARIABLES
int status;
int counter;
int new_byte;
int valid;
int output;
int prod;
int tmr_counter;
int data[3]; // array of 3 2-byte sample values
int *data_ptr;
int smp_count;
int tmr_sample;

//MAIN
#pragma code
void main(void) {

        // Initialize every value used by the interrupt in case it
        // triggers early
        data_ptr = &data[0];
        tmr_counter=0x0000;
        smp_count = 0;
```

```
//Configure PIC transmit and receive for full-duplex mode
TXSTA = 0x24;           // Async mode, transmit enabled, high Baud
RCSTA = 0x90;           // Async mode, receiver and serial port enabled
SPBRGH = 0x00;
SPBRG = 0x09;           // Baud rate is 125kHz
TRISD = 0xFF; // PORTD is input
TRISC = 0xF0; // Top nybble is input, bottome nybble is output
TRISB = 0x00; // PORTB is output
ADCON1 = 0xFF;          // analog to digital
INTCON = 0xC0;          // Interrupts enabled
PIE1 = 0x20;    // Enable recieve interrupts
T0CON = 0x08;           // Leave off, but initialize anyway.

//Initialize more values
PORTD = 0x00;
status = 0x00;
LATBbits.LATB5 = 0;


//RC5 is an output from the FPGA, toggled when new byte sent
new_byte = PORTCbits.RC5;
while(1){
        //RECEIVING BYTES FROM THE FPGA
        if(status==0x90){ // Note on message for ch0 is 0x90
                valid = 1;
        }
        else if (status==0x80){ // Note off message is 0x80
                valid = 2;
        }
        else{           // All other MIDI messages are ignored
                valid = 0;
        }
        while((new_byte==PORTCbits.RC5)&(INTCONbits.TMR0IF==0)){
        } // While there's no MIDI bytes or samples in memory, wait
        if (new_byte !=PORTCbits.RC5){  // new MIDI byte
                new_byte=PORTCbits.RC5; // reset new_byte indicator
                if(PORTDbits.RD7==1){   // Status bytes have MSB=1
                        counter = 0;
                        status = PORTD; // Save the status byte
                }
                else{                   // Data bytes have MSB=0
                        if(counter==0){
                                if(valid==1){ // Note on, send note
                                        TXREG = PORTD;
                                        counter = counter+1;
                                }
                                else if(valid==2){ // Note off,
                                                   // send 0x00
                                        TXREG = 0x00;
                                        counter = counter+1;
                                }
```

```
                        }
                    }
                }

                //OUTPUTTING SAMPLES
                //Run a timer at 3kHz. Output samples when timer goes off.
                //Output three, then turn timer off. Interrupt turns timer
                //on when LabView sends the next three bytes.
                else if(INTCONbits.TMR0IF){ // TMR0 has overflowed
                    INTCONbits.TMR0IF = 0; // Clear flag bit
                    if (tmr_counter == 2){// Stop timer when all sent
                        T0CON = 0x08;
                    }
                    TMR0H = 0xFE; // Reinitialize the timer.
                    TMR0L = 0x88; // Value determined empirically.
                    tmr_sample = *(data_ptr+tmr_counter); //Get sample
                    output = tmr_sample+0x80; // Remove discontinuity
                                              // from 2's compliment
                    PORTC = output;             // Output the lower 4 bits
                    LATBbits.LATB5 = 1; // Toggle a control signal
                                        // for the FPGA's de-serializer
                    output = output>>4; // Retrieve upper nybble
                    PORTC = output;     // Output the upper nybble
                    LATBbits.LATB5 = 0; // Toggle the control signal
                    if (tmr_counter<2){
                        // increment array index
                        tmr_counter = tmr_counter + 1;
                    }
                    else{
                        tmr_counter = 0; // reset array index
                    }
                }
            }
        }

//INTERRUPT
#pragma interrupt isr     // Tells compiler to preserve all memory
void isr(void) {
        PIR1bits.RCIF=0;                    // Clear the interrupt flag
        *(data_ptr+smp_count) = RCREG;  // Store the sample in memory
        if (smp_count<2){                   // If not on last sample
            smp_count = smp_count + 1; // increment array index
        }
        else{ // if received the last sample, reset the array index
            smp_count = 0;
            tmr_counter = 0;  // tmr_counter should also be zero
            TMR0H = 0xFF;
            TMR0L = 0xFF;           // reinitialize timer
            T0CON = 0x81;           // Restart timer
        }
}
```

**Appendix D:**
Main while loop of the Labview VI.