Lab 3 Report - Analog Interfaces

Sahil Bissessur, Vincent "Styxx" Chang, Enrique Gutierrez
TA: Carrie Segal, W @ 4PM

Objective

In this lab, we work with digital-to-analog converters (DACs) and analog-to-digital converters (ADCs) to understand analog interfaces better.

- In Part 1, we use a DAC to create a triangle wave and a sine wave with variable frequency and amplitude by coding the workings of the DAC in Verilog and coding the user interface in C.
- In Part 2, we use an ADC to save 30 seconds of audio analog input data and then connect the DAC to play back the saved audio, allowing for multiple modes of playback.

Design Methodology

Overview

In part 1, we are tasked with creating a triangle wave and a sine wave, both with the ability to change frequencies and amplitudes through an interface similar to the one we used in Lab 2 by running `e2.out` in SPX. By wiring the DAC accordingly on the breadboard with – with its according ±5V and +12V voltages – and connecting the data wires on the DAC to specific output pins on the CPLD, we effectively connect the DAC to the DSP through the CPLD.

In the Verilog, given a specific address (0x10dXX) being written to on the address lines, we latch the data bus from the DSP to the data lines on the DAC, which then outputs an analog value between $-V_{REF}$ (-5V) and $(127/128)V_{REF}$ depending on the binary value (from 0 to 255).

Our C code is created as the interface between the user and the DSP. The C code takes in a character value from standard in and, depending on the value:

- Changes the wave to a sine wave, a triangle wave, or a combination of a sine+triangle wave
- Changes the frequency of the wave
- Changes the amplitude of the wave

The C code handles creating the data bits for the DAC – with a counter for the triangle wave and a look-up table for the sine wave – as well as the changes in frequency and amplitude and handling the user interface, while the Verilog code simply latches on the data values and sends them to the DAC.

With this combination of C and Verilog, our DAC should effectively output multiple waveforms with multiple attributes cleanly and efficiently through the coded interface.

In part 2, we are tasked with recording 30 seconds of audio with our ADC and storing it then replaying it with our DAC by offering multiple methods of playback. While we were unable to fully create a design for this portion of the lab, we surmise it would run similar to part 1, except in reverse:

After wiring the ADC on the breadboard – with its according capacitors and resistors – we would attach the CLK, D_{out}, D_{in}, and CS* pins of the ADC to the CPLD.

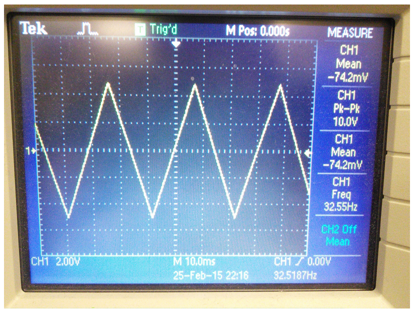
In C, we would program a similar, but simpler, interface that would allow data to be written to the DSP from the ADC by accepting write cycles to addresses 0x10cXX0 and 0x10cXX1, then allowing a read cycle from address 0x10ccXXX to send the most recent converted data, which would be handled in Verilog by the CPLD, all while ensuring that we met the different clock requirements by the chip and the DSP.

However, we were confused as to where to store this data and how to access this portion of memory for playback on the DSP. Not only this, but how to add the necessary playback effects with the DAC.

Results

Metric results

While we were able to create a stable triangle wave for Part 1, we were unable to change its frequency or amplitude. We also were not able to progress enough to create a sin wave or create a working interface in C (due to our large complications with Code Composer not working as intended – something that was out of our control). Because of this, we were unable to fully finish Part 1 or begin Part 2; however, we have skeleton code for all of part 1 that – with some more time and refinement – would definitely work in completing the necessary requirements for part 1.



Picture 1: Stable Triangle Wave

Design Results

What was outlined above in *Design Methodology* was our intended way of handling Part 1 in Lab 3. However, due to complications with Code Composer, we were required to modify our design such that instead of C handling the majority of the code, the Verilog handles it.

In the Verilog, given a specific address (`0x10aXXX` to `0x10fXXX`) being written to on the address lines, we do the following things:

- `0x10a`: Switches the wave to a sine wave
- `0x10b`: Switches the wave to a triangle wave
- `0x10c`: Left shifts a bit 0 into the amplitude variable (changing the amplitude)
- `0x10d`: Left shifts a bit 1 into the amplitude variable (changing the amplitude)
- `0x10e`: Left shifts a bit 0 into the period variable (changing the frequency)
- `0x10f`: Left shifts a bit 1 into the period variable (changing the frequency)

We could also add in writing into address `0x101` to switch the wave to a combination of the sine/triangle wave.

If the address is in the latter four cases, the respective variable would simply change. If the address was in the previous two cases, the finite state machine would shift to the according state and output a sine wave or a triangle wave with the according attributes given the amplitude and period variables.

Working off `sample.c`, our C code interface is implemented such that there are two void functions that get called within `void c_int99()`. These functions are called `triangle()` and `sinewave()`; they pass an integer zero stored in `int value` to the address 0x10b000 or 0x10a000 respectively, which would be read by our Verilog and to start implementing the desired waveform. Ideally we would only write to one address (0x10d000) and our C code would handle if it was a triangle wave or a sine wave; however, because we couldn't get the compiler to work, we implemented it this way so that we could still test our Verilog code.

The main function is where we take the user input: we prompt the user to tell us the kind of waveform they want as well as the desired amplitude and frequency. Then we check what kind of waveform the user wants, and call the function `c_int99()` in a continuous while loop. Within the function `c_int99()`, we check what the user input was again; if it is a sine wave we call the `sinewave()` function and if its a triangle wave we call the `triangle()` function. In `c_int99()` we also check if the amplitude and frequency are in range. If they are out of bounds, we say print out an error within the main function; otherwise we check every bit of the amplitude and the frequency and left shift those values into a integer. Every time we left shift a 1-bit we write to 0x10d00 for amplitude and 0x10f000 for frequency, while every time we left a 0-bit, we write to 0x10c000 for amplitude and 0x10e000 for frequency. The Verilog code takes care of the shifting once we write to these specific addresses.

The problem with this implementation is that the ispLever compiler – for whatever reason – is incredibly finicky to work with in code. While it makes these assumptions and collapses down nodes that it believes is not being used, this tends to result in more complications with things like Constraint Editor portion or the execution of the code than expected. Also, the compiler seems to be outdated such that it does not support certain syntaxes within Verilog; this often limited our abilities to handle the design – as it has in previous labs. In this lab, it troubled our implementation for the look-up table, which seriously impeded our ability to produce a sine wave.

Because we were unable to finish Part 1 completely, we were also unable to start Part 2.

Road Blocks and Parts That did not Meet Spec

While designing for the DAC, we originally built our Verilog such that it simply took inputs from the CPLD – given the correct address and a low strobe signal – and latched the data values to send them to the DAC chip. We intended for the C code to handle sending these data values from the DSP to CPLD; this is the implementation most of the people used.

However, when we finished our Verilog code and attempted to move onto C, Code Composer would simply not compile our file or build the project. We followed all of the instructions for setup just as in lab 1 – multiple times, sinking an unnecessary amount of hours into this very, very odd problem – but we were simply unable to get Code Composer to build any sort of snippet of code at all.

Because of this, we were unable to create the .out file from `sample.c` to understand how to build the interface required in order to work the DAC. Not only that, but because we spent a large amount of time on a silly problem that should never have happened (such that other people found a solution by "switching to another computer" or "logging into another account" - neither of which worked for us for whatever reason) we had to completely change our design, implementing everything in Verilog with minimal work being done by the interface in C. This left us with a very small amount of time to complete the lab in this different design.

The current design for part 1 does not take into account the creation of a triangle wave with variable attributes or a sine wave. Because the ispLever compiler is outdated such that our design to initialize a look-up table in Verilog was not accepted by the compiler – it simply didn't support the format of initialization of the array – we were unable to use a look-up-table to produce our sine wave. All other attempts to utilize this look-up table was also rejected by the compiler – often due to "not having enough memory" or something along those lines. Because of our limited time in working with this lab due to our problems with Code Composer, we were unable to refine our design for part 1 to include variable amplitude and frequency or complete the creation of a sine wave.

Issues and Sources of Errors

We're confident that had Code Composer worked as intended such that we would not have spent too many hours trying to fix the issue, we would have been able to successfully complete all of Part 1 in the lab. It's frustrating to be hit with a road block that is dependent on something as arbitrary as the terminal you are using when it clearly was a complete non-issue for a large majority of the other groups and, because of that, have to create a design with limited capabilities due to the limited amount of hours we had after having to work around it. That's the only issue we have with the lab, since we were unable to reach Part 2.

Source Code

interface.c – C interface

```
#include <stdio.h>
#include <math.h>

#define BRANCH_TO(x) (0x60000000 | (int)(x))
#define at(x,y) printf("\33[%d;%dH",(y),(x))

void c_int99();             /* timer 1 interrupt routine */

int mytime,starttime;       /* 100Hz timecount & startcnt*/
int period;                 /* used in timer period setup*/
int x=0;
float frequency;
int value = 0;
int amp;
char shape[50];
int invalid_amp;
int invalid_freq;
int c1,k1,c2,k2;

/* macro to write lower 16-bits of 'x' to LEDs           */
#define LEDS(x) *(volatile int*)0x101000 = (x)

/* address of timer #1 control register array           */
#define TIMER1  ((volatile int*)0x808030)

void set_timer1(int period)   /* to set timer#1 period  */
{
    TIMER1[0] = 0x203;      /* hold timer, choose options  */
    TIMER1[8] = period;     /* set T1 period in H1/2 cycles*/
    TIMER1[0] = 0x2c3;      /* make it run                 */
}

 // Our triangle wave function
void triangle() {
   // Write value to register
   *(volatile int*)0x10b000 = value;
}

void sinewave(){
    *(volatile int*)0x10a000 = value;
    }

void c_int99()             /* timer interrupt routine*/
{
    asm("  push dp");
    asm("  ldp _mytime"); /* ensure data addressibility */
    mytime++;

/* whatever you want here ... but no printf() or other std I/O   */
/* The amount of code in an interrupt routine should be minimal. */
    LEDS( mytime );
    asm("  pop dp");

   if (shape[0] == 's'){sinewave();}
```

```
    if (shape[0] == 't'){triangle();}

    if ((amp < 0) || (amp >= 6)) { invalid_amp=1; }
     else {
          invalid_amp=0;
          for (c1 = 2; c1 >= 0; c1--) {   // 3 bit reg
                k1 = amp >> c1;
                if(k1 & 1) { *(volatile int*) 0x10d000 = 0; } // Write to 10d - left
   shift 1
                else { *(volatile int*) 0x10c000 = 0; }// Write to 10c-left shift 0
          }
    }
   if ((freq < 0) || (freq > 10000)) {
    invalid_freq=1;
   }
   else {
          invalid_freq=0;
          for (c2 = 31; c2 >= 0; c2--) {         // 32 bit reg
                k2 = freq >> c2;
                if (k2 & 1) {*(volatile int*) 0x10f000 = 0;}//Write to 10f-left
   shift 1
                else { *(volatile int*) 0x10e000 = 0; }// Write to 10e-left shift 0
          }
    }

   }

}
main() {
   char line[100];

   mytime  = 0;
   period  = 10000;     //default value of 10000

    printf("Please enter desired wave type (triangle or sine): ");
   scanf("%s", &shape);

    printf("Please enter desired amplitude: ");
   scanf("%d", &amp);

   if(invalid_amp=1){
         printf("Invalid amplitude");
   }
     printf("Please enter desired frequency: ");
     scanf("%f", &frequency);

     if(invalid_freq=1){
         printf("Invalid frequency\n");
     }
   /* set timer to run at 5M/period interrupts per sec  */
   set_timer1(period);

   *(int*)0x809fca = BRANCH_TO(c_int99); /* install TINT1 vector */

   /* Enable bus transceivers */
   *(volatile int*)0x104001 = 1;


   if (shape[0] == 't'){    /* if triangle wave is selected */
```

```
        while(1) {

              /* Run triangle wave function */
            //set_timer1(period);
              c_int99();

        }  }
   if (shape[0] == 's'){    /* if sine wave was selected */
        while(1){
        c_int99(); }
        }

    /* Disable transceivers */
   *(int*)0x104000 = 1;

    asm(" ldi 0,ie");             /* mask ALL interrupts      */
   asm(" ldi 1,if");             /* select proper boot space */
   asm(" br  45h");              /* enter boot loader        */

   exit(0);    /* not needed .... we are booting back TO SPX */
}
```

DAC.v – Verilog Handler

```
/*************************************************************/
/*  DAC verilog                                              */
/*  Sahil Bissessur, Vincent "Styxx" Chang, Enrique Gutierrez  */
/*************************************************************/

module DAC(add, clk, strobe, Dout);

      input wire [11:0] add;
      input wire clk;
      input wire strobe;

      output reg [7:0] Dout;

      reg [7:0] Din = 8'b00000000;

      reg up = 1;             // If increasing or decreasing in tri_latch
      reg [7:0] arb = 0;      // Increment-er through sine LUT

      // To change the counter, we left shift into it.
      // This will then change the frequency of the wave.
      // Larger counter = lower frequency.
      // Need to find formula to make this work such that given a frequency,
      // counter changes to correct value (DSP runs on 10Mhz)
      reg [31:0] counter = 0;
      reg [31:0] period = 32'd200;

      // To change the amplitude, we left shift into it
      // This will then change the amplitude of the wave (MAX is 5)
      // Make it so that Dout = Din/Amplitude?
      // Or have amplitude be a percentage and we multiply that by Din?
      reg [2:0] amplitude = 3'b001; // this needs to be changeable


      //http://jacaheyo.blogspot.com/2012/06/implementing-lut-in-verilog.html
      reg [7:0] LUT [0:255] = {8'd128,8'd131,8'd134,8'd137,8'd141,8'd144,8'd147,
      8'd150,8'd153,8'd156,8'd159,8'd162,8'd165,8'd168,8'd171,8'd174,8'd177,8'd180,
      8'd183,8'd186,8'd188,8'd191,8'd194,8'd196,8'd199,8'd202,8'd204,8'd207,8'd209,
      8'd212,8'd214,8'd216,8'd219,8'd221,8'd223,8'd225,8'd227,8'd229,8'd231,8'd233,
      8'd234,8'd236,8'd238,8'd239,8'd241,8'd242,8'd244,8'd245,8'd246,8'd247,8'd249,
      8'd250,8'd250,8'd251,8'd252,8'd253,8'd254,8'd254,8'd255,8'd255,8'd255,8'd256,
      8'd256,8'd256,8'd256,8'd256,8'd256,8'd256,8'd255,8'd255,8'd255,8'd254,8'd254,
      8'd253,8'd252,8'd251,8'd250,8'd250,8'd249,8'd247,8'd246,8'd245,8'd244,8'd242,
      8'd241,8'd239,8'd238,8'd236,8'd234,8'd233,8'd231,8'd229,8'd227,8'd225,8'd223,
      8'd221,8'd219,8'd216,8'd214,8'd212,8'd209,8'd207,8'd204,8'd202,8'd199,8'd196,
      8'd194,8'd191,8'd188,8'd186,8'd183,8'd180,8'd177,8'd174,8'd171,8'd168,8'd165,
      8'd162,8'd159,8'd156,8'd153,8'd150,8'd147,8'd144,8'd141,8'd137,8'd134,8'd131,
      8'd128,8'd125,8'd122,8'd119,8'd115,8'd112,8'd109,8'd106,8'd103,8'd100,8'd97,
8'd94,8'd91,8'd88,8'd85,8'd82,8'd79,8'd76,8'd73,8'd70,8'd68,8'd65,8'd62,8'd60,
8'd57,8'd54,8'd52,8'd49,8'd47,8'd44,8'd42,8'd40,8'd37,8'd35,8'd33,8'd31,8'd29,
      8'd27,8'd25,8'd23,8'd22,8'd20,8'd18,8'd17,8'd15,8'd14,8'd12,8'd11,8'd10,8'd9,
      8'd7,8'd6,8'd6,8'd5,8'd4,8'd3,8'd2,8'd2,8'd1,8'd1,8'd1,8'd0,8'd0,8'd0,8'd0,
      8'd0,8'd0,8'd0,8'd1,8'd1,8'd1,8'd2,8'd2,8'd3,8'd4,8'd5,8'd6,8'd6,8'd7,8'd9,
8'd10,8'd11,8'd12,8'd14,8'd15,8'd17,8'd18,8'd20,8'd22,8'd23,8'd25,8'd27,8'd29,
8'd31,8'd33,8'd35,8'd37,8'd40,8'd42,8'd44,8'd47,8'd49,8'd52,8'd54,8'd57,8'd60,
8'd62,8'd65,8'd68,8'd70,8'd73,8'd76,8'd79,8'd82,8'd85,8'd88,8'd91,8'd94,8'd97,
      8'd100,8'd103,8'd106,8'd109,8'd112,8'd115,8'd119,8'd122,8'd125};
      //reg [7:0] LUT [0:255];
```

```
	//LUT[0] = 8'd128;

	// If you need to add more states, then don't forget to increase
	// the bit count if necessary
	parameter starting =	2'b00;
	parameter tri_latch =	2'b01;
	parameter sine_latch =	2'b10;
	parameter latch =	2'b11;

	// If you're only testing a specific latch, set it to that latch
	// Otherwise, if you're testing the entire thing, set it to starting
	reg [1:0] state = starting;

	always @ (negedge clk) begin
		case (state)
			// Starting idle state
			// Checks for addresses and executes according action
			starting:
				begin
					// Next State
					if ((add[11:0] == 12'h10a) && (~strobe)) begin
						state <= sine_latch;
					end
					if ((add[11:0] == 12'h10b) && (~strobe)) begin
						state <= tri_latch;
					end
					if ((add[11:0] == 12'h10c) && (~strobe)) begin
						amplitude <= amplitude << 1;
						amplitude[0] <= 0;

					end
					if ((add[11:0] == 12'h10d) && (~strobe)) begin
						amplitude <= amplitude << 1;
						amplitude[0] <= 1;
					end

					if ((add[11:0] == 12'h10e) && (~strobe)) begin
						period <= period << 1;
						period[0] <= 0;
					end
					/* //Unknown as to why inserting this segment of code
					   //ruins the JEDEC file and ruins compiliation
					if ((add[11:0] == 12'h10f) && (~strobe)) begin
						period <= period << 1;
						period[0] <= 1;
					end*/

				end
			// Tri_latch
			// Outputs a triangle wave
			// Currently disregards amplitude changes
			tri_latch:
				begin
					// Output
					if (counter == 200) begin		// Smaller counter
= higher frequency
						counter <= 0;
						if (up == 1) begin
							Din <= Din + 1;
```

```
                            if (Din == 8'b11111110) begin
                                up <= 0;
                            end
                        end
                        else begin
                            Din <= Din - 1;
                            if (Din == 8'b00000001) begin
                                up <= 1;
                            end
                        end

                    end
                    else begin
                        counter <= counter + 1;
                    end
                    // Next State
                    state <= latch;
                end

            // Sine latch
            // Outputs a sin wave
            // Currently disregards amplitude changes
            sine_latch:
                begin
                    // Sine wave
                    if (counter == period) begin
                        counter <= 0;
                        Din <= LUT[arb];
                        arb <= arb + 1;
                    end
                    else begin
                        counter <= counter + 1;
                    end

                    // Next State
                    state <= latch;
                end

            // Latch State
            // Latches the value of Din to Dout
            latch:
                begin
                    // Output
                    Dout <= Din;

                    // Next State
                    // If you're only testing a specific latch, set it to
that latch
                    // Otherwise, if you're testing the entire thing, set
it to starting
                    state <= starting;
                end
        endcase
    end

endmodule
```