

Neuron Project

EE-2902 Spring 2011

Colin Stapleton

5/19/2011

Table of Contents

Introduction	3
Explanation of the Project	3
Top Level	3
Audio Codec Controller	3
Delay Counter	6
ADC DAC Controller.....	6
Neuron Model.....	7
Conclusions	8
References	9
VHDL Files.....	9
Top Level – Neuron_Project.vhd.....	9
audioCodecController.vhd	13
delayCounter.vhd.....	16
bclk_counter.vhd	18
LRchannelCounter.vhd.....	18
AdcDacController.vhd	19
neuronModel.vhd	21
signedMultiplier.vhd	23
ROMcontroller.vhd	24
dataBitCounter.vhd.....	25
clock50KHz.vhd	26

Introduction

The goal of this project was to simulate neuron-firing activity on an oscilloscope, using the Cyclone II FPGA on an Altera DE1 board. The neuron firing was simulated using Euler's method, and the simulation data was sent to the oscilloscope via the digital to analog codec on the DE1. In order to use the Wolfson codec on the DE1, the codec's control registers had to be initialized. The codec was initialized using the two-wire I²C protocol, between the FPGA and the codec. The initialization data was stored in a ROM module in the FPGA.

Explanation of the Project

The neuron project was divided into modules, which were designed and tested individually. The top level connected each of the modules, to run the simulation. Each module was created in VHDL, and synthesized using Altera's Quartus II development software. The modules were tested using Mentor Graphic's ModelSim waveform simulation tool.

Top Level

The inputs to the top level of the project were a button on the DE1, the 50MHz and 27MHz clocks on the FPGA, two switches on the DE1, and the analog line-in audio input on the DE1. The button was pressed to reset the modules in the project to their starting conditions. The two switches were used to select the output: neuron model V, neuron model U or a loop-back test for the audio codec.

The 50MHz and 27MHz clocks were used to clock the synchronous modules of the project. The 50MHz clock was passed through one of the four phase-locked loops (PLL) on the FPGA, producing a second 50MHz signal. The 27MHz clock was passed through another of the four PLLs to produce an 18.42105MHz clock for use by the audio codec.

The top level connected the following modules: the Audio Codec Controller, Delay Counter, ADC DAC Controller, and Neuron Model. The top level also controlled a tri-state buffer on the I²C data line. When sending data to the audio codec from the FPGA, the tri-state buffer was enabled. When waiting for an acknowledgement from the codec, the tri-state buffer was set to high impedance. When the tri-state buffer was set to high impedance, pull-up resistors on the I²C data line pulled the line high. If the audio codec acknowledge the sent data, it pulled the line low instead.

Audio Codec Controller

The audio codec controller initialized the Wolfson audio codec by sending a series of 10 data packets via an I²C interface. Each data packet was 24 bits long, consisting of: the codec address, the control register, and the control register settings.

The 10 packets were retrieved from ROM, and sent over the two-wire interface via a finite state machine (FSM). The FSM contained 6 states: reset, start condition, send data, acknowledge, prepare for stop condition, and stop condition. The FSM switched to the reset state when the reset button on the DE1 was pressed. Following reset, the FSM entered the start condition, in which the I²C clock line was held high while the I²C data line was pulled low. The FSM then entered the send data state, where each of the 24 bits of the current packet were sent. After each 8 bits, the FSM entered the acknowledge state, where the I²C data line was set to high impedance. Following transmission of the full 24 bit packet, the FSM entered the stop condition, where the I²C clock line was held high, while the I²C data line was pulled high.

The I²C clock signal was a 50KHz signal, generated from the 50MHz FPGA clock using a counter. In order to coordinate between the FSM placing each data bit on the I²C data line, and sending the corresponding I²C clock pulse, two 50KHz clocks were used. The I²C clock lagged the FSM clock by a half-period. This allowed the FSM to stabilize each bit of data, prior to the bit being read by the codec.

Figure 1 shows a ModelSim simulation of the audio codec controller sending the 10 packets.



Figure 1: Audio codec controller sending 10 24-bit data packets to the audio codec via I²C.

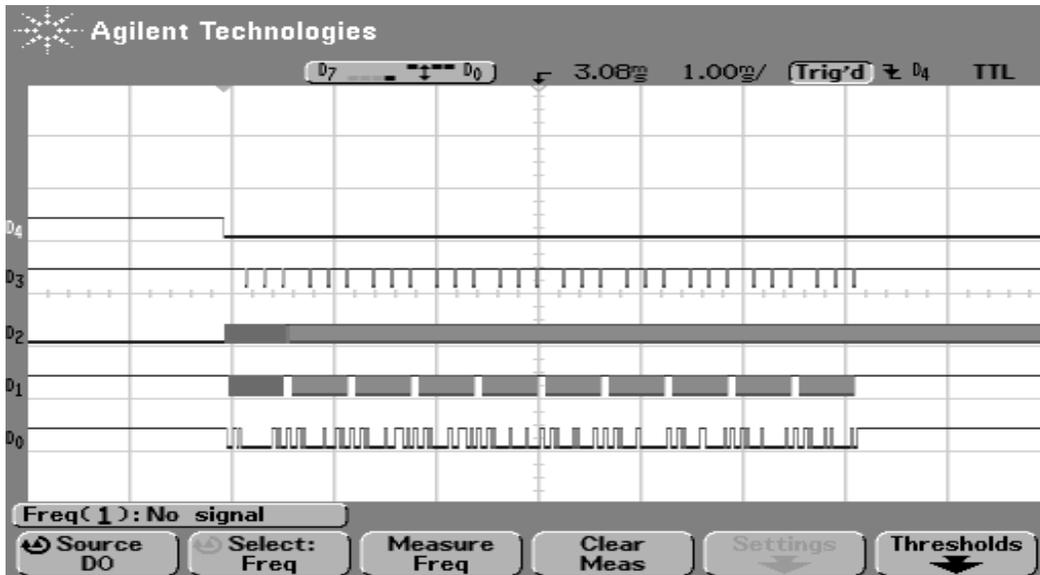


Figure 2: 10x24 bit data transfer on the actual I²C busses, as measured by logic analyzer. From top to bottom, the lines are: active-high reset, tri-state buffer control, FSM 50KHz clock, I²C 50 KHz clock, I²C data.

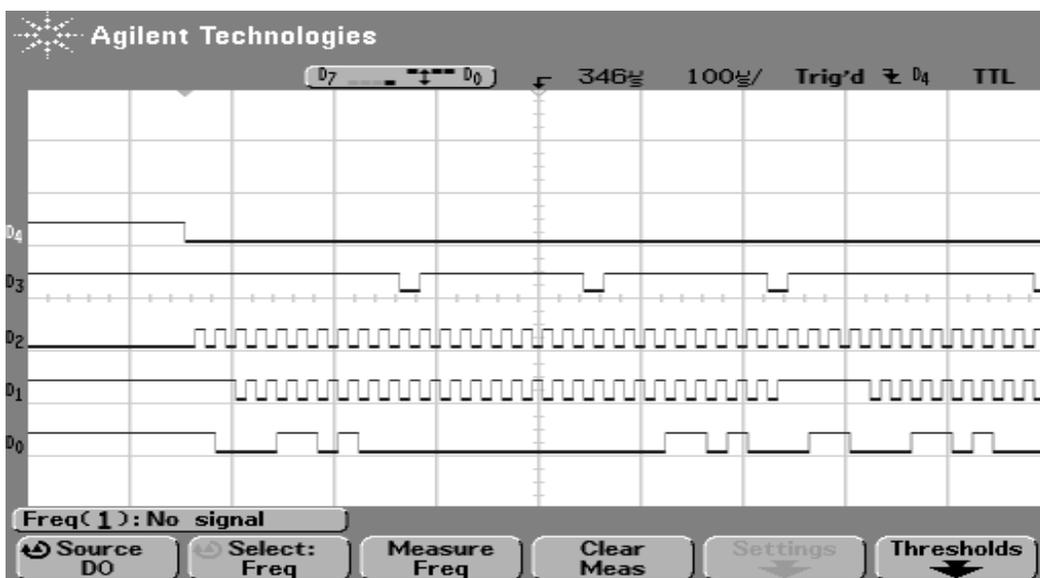


Figure 3: Zoomed-in view of the first 24 bit data packet sent to the audio codec.

Delay Counter

To allow time for the audio codec to be initialized, a 40ms delay was initiated following reset. At the end of the delay, a start signal was sent to the ADC DAC controller module.

ADC DAC Controller

The ADC DAC controller module remained in reset until 40ms after the reset button was pressed on the DE1. The controller then generated two clock signals, which were used by the codec. The bit clock signal was used to clock the output bits from the digital to analog conversion by the codec. The bit clock signal ran at 3.07MHz, using the 18.42105MHz signal produced at the top level. A second clock, the left/right select clock, ran at 192KHz. The left/right select clock was used to switch channels within the codec.

The ADC DAC controller used the two switch inputs from the DE1 to select one of three possible outputs: neuron V, neuron U, and loop-back. The loop-back test passed analog data from the line-in jack on the DE1, through the codec, and out the line-out jack.

The neuron V and U model data was passed into the ADC DAC controller, and sent out, one bit at a time, through the codec DAC using the bit clock.

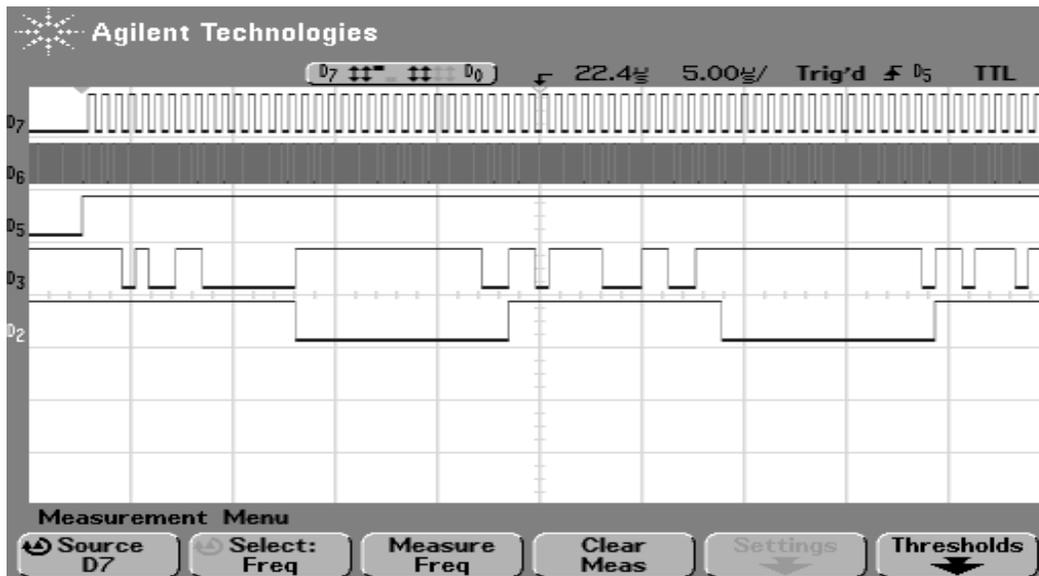


Figure 4: ADC DAC controller signals, top to bottom: bit clock, 18.42105MHz clock, active-low reset, codec data, left/right channel select.

Neuron Model

The neuron model used Euler's method to generate simulated firings of neurons. The model used a fixed-point multiplier to perform the needed arithmetic, and produced two output signals, V and U.

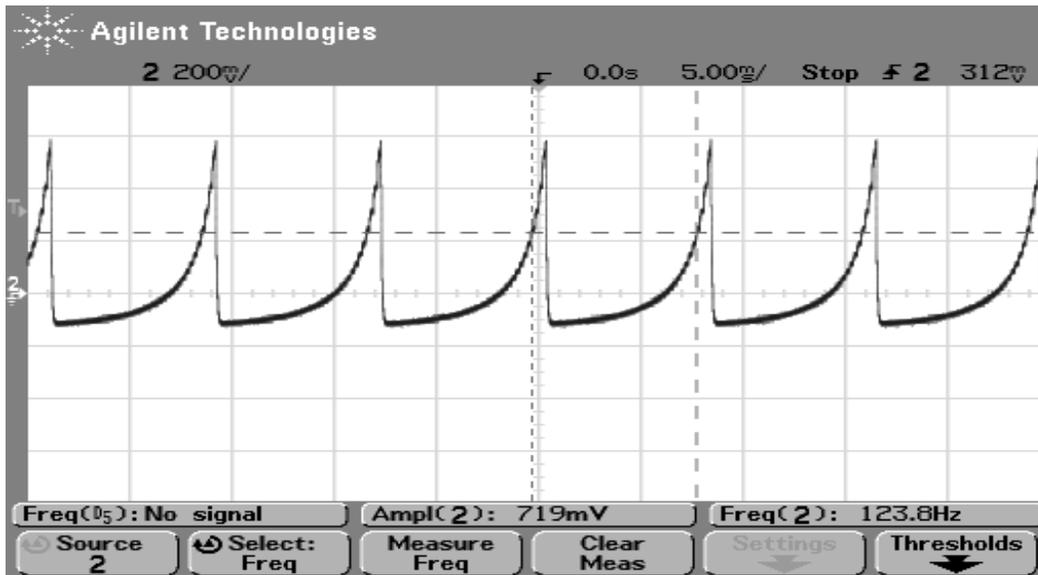


Figure 5: Neuron V output analog signal.

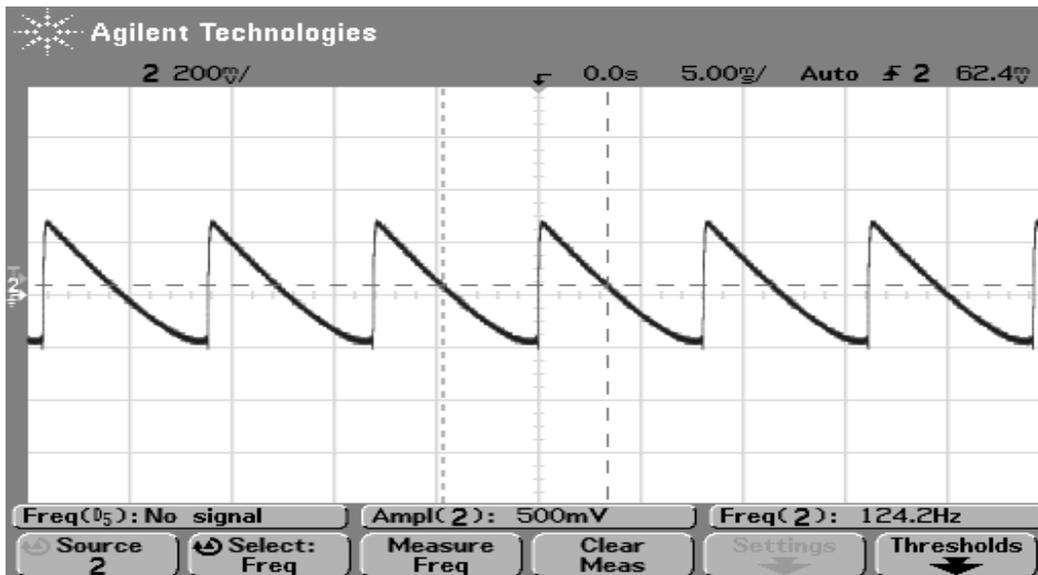


Figure 6: Neuron U output analog signal.

Conclusions

This project demonstrated coordination between the FPGA and the following peripheral devices: phase-locked loops, read-only memory, and an ADC DAC codec. The project also demonstrated the successful use of the I²C protocol to send data, and the creation of a fixed-point multiplier and clock dividers. The project was successful in generating both types of analog neuron signals on the line-out jack of the DE1.

References

1. EE 2902 Neuron Project Specifications created by Dr. Bharathwaj Muthuswamy.
Wolfson audio codec specifications.
<http://myweb.msoe.edu/muthuswamy/DigitalSystemsDesign/ee2902/project/>
2. Cornell University FPGA neuron model.
<http://people.ece.cornell.edu/land/courses/ece5760/DDA/NeuronIndex.htm>

VHDL Files

Top Level – Neuron_Project.vhd

--4/15/11
--Complete neuron simulation program.
--Produces neuron-firing waveforms on an oscilloscope using Euler's method.

LIBRARY ieee;
USE ieee.std_logic_1164.all;

```
entity Neuron_Project is
    port(
        KEY: in std_logic_vector(3 downto 0);
        CLOCK_50: in std_logic;
        CLOCK_27: in std_logic_vector(1 downto 0);
        --I2C ports
        I2C_SCLK: out std_logic;
        I2C_SDAT: inout std_logic;
        --audio codec ports
        AUD_ADCLDAT: in std_logic;
        AUD_ADCLRCK: out std_logic;
        AUD_DACLCK: out std_logic;
        AUD_DACDAT: out std_logic;
        AUD_XCK: out std_logic;
        AUD_BCLK: out std_logic;
        --select loopback test or neuron model output
        SW: in std_logic_vector(9 downto 0);
        --output for logic analyzer testing
        GPIO_0: inout std_logic_vector (35 downto 0);
        LEDG: out std_logic_vector (7 downto 0)
    );
end Neuron_Project;
```

```
architecture toplevel of Neuron_Project is
```

```
    --PLL from MegaWizard in Quartus.
    --both input and output are 50MHz
```

```

component clockBuffer IS
  PORT
  (
    areset      : IN STD_LOGIC := '0';
    inclk0      : IN STD_LOGIC := '0';
    c0          : OUT STD_LOGIC
  );
END component;

--18.42105 MHz master clock (should be 18.432MHz, but this is the closest the PLL can get)
component audioPLLClock IS
  PORT
  (
    --active high reset
    areset      : IN STD_LOGIC := '0';
    inclk0      : IN STD_LOGIC := '0';
    c0          : OUT STD_LOGIC
  );
END component;

--I2C controller to drive the Wolfson codec
component audioCodecController is
port(
  clock50MHz,reset: in std_logic;
  I2C_SCLK_Internal: out std_logic;
  --must be inout to allow FPGA to read the ack bit
  I2C_SDAT_Internal: out std_logic;
  SDAT_Control: out std_logic;
  --for testing
  clock50KHz_Out: out std_logic
);
end component;

--waits 40ms, then asserts high output
component delayCounter is
  port(
    clock,reset: in std_logic;
    --active high reset
    resetAdc: out std_logic
  );
end component;

--generates digital audio interface clock signals
--starts after delayCounter asserts (40ms)
component AdcDacController is
  port(
    --reset signal starts '0', then goes to '1' after 40 ms => active-low
    resetn: in std_logic;
    --from 50MHz PLL at toplevel
    clock18MHz_in: in std_logic;
    --line-in on the DE1

```

```

        adcData: in std_logic;
        --line-out on the DE1
        dacData: out std_logic;
        bitClock: out std_logic;
        dacLRSelect: out std_logic;
        adcLRSelect: out std_logic;
        --neuron model signals
        neuronVin: in std_logic_vector(15 downto 0);
        neuronUin: in std_logic_vector(15 downto 0);
        --selects loopback or neuron signals using switches
        outputSelect: in std_logic_vector(1 downto 0)
    );
end component;

--generates the neuron model signals using euler's method
component neuronModel is
    port(
        frameClock: in std_logic;
        resetn: in std_logic;
        vOut: out std_logic_vector(17 downto 0);
        uOut: out std_logic_vector(17 downto 0)
    );
end component;

--clock signal from the PLL clockBuffers
signal clock50MHz: std_logic;

--18MHz PLL output signal
signal clock18MHz: std_logic;

--asynchronous reset for the whole project
signal reset: std_logic;

--I2C data and clock lines
signal i2cData, i2cClock: std_logic;

--tri-state buffer control
signal i2cDataControl: std_logic;
signal i2cDataTriState: std_logic;

--assert signal from delay counter
signal codecResetn: std_logic;

--audio codec signals
signal adcDat_sig: std_logic;
signal adcLRCK_sig: std_logic;
signal dacLRCK_sig: std_logic;
signal dacDat_sig: std_logic;
signal bck_sig: std_logic;

--neuron model signals

```

```
signal vNeuronSig: std_logic_vector(17 downto 0);
signal uNeuronSig: std_logic_vector(17 downto 0);
signal outputSelect: std_logic_vector(1 downto 0);
```

```
--for testing
signal clock50KHz: std_logic;
```

```
begin
```

```
--keys are active low
reset <= not KEY(0);
```

```
--selects loopback test or neuron model output
outputSelect <= SW(1 downto 0);
```

```
--PLLs
clockBufferInstance: clockBuffer port map(reset,CLOCK_50,clock50MHz);
audioPLLClockMap: audioPLLClock port map(reset, CLOCK_27(0), clock18MHz);
```

```
--I2C
I2CControllerInstance: audioCodecController port map(clock50MHz, reset, i2cClock, i2cData,
    i2cDataControl, clock50KHz);
```

```
--Delay counter
delayCounterMap: delayCounter port map(clock50MHz, reset, codecResetn);
```

```
--Codec Controller
AdcDacControllerMap: AdcDacController port map(codecResetn, clock18MHz, adcDat_sig, dacDat_sig,
    bck_sig, dacLRCK_sig, adcLRCK_sig, vNeuronSig(17 downto 2), uNeuronSig(17 downto 2), outputSelect);
```

```
--neuron model
neuronModelMap: neuronModel port map(adcLRCK_sig, codecResetn, vNeuronSig, uNeuronSig);
```

```
--tri-state data output
i2cDataTriState <= i2cData when i2cDataControl = '1' else 'Z';
```

```
--I2C output ports
I2C_SCLK <= i2cClock;
I2C_SDAT <= i2cDataTriState;
```

```
--audio codec input port
adcDat_sig <= AUD_ADCDATA;
```

```
--audio codec output ports
AUD_ADCLRCK <= adcLRCK_sig;
AUD_DACLCK <= dacLRCK_sig;
AUD_DACDAT <= dacDat_sig;
AUD_XCK <= clock18MHz;
AUD_BCLK <= bck_sig;
```

```
--for testing
```

```

-- GPIO_0(0) <= I2C_SDAT; --only this one is tri-state
-- GPIO_0(1) <= i2cClock;
-- GPIO_0(2) <= clock50KHz;
-- GPIO_0(3) <= i2cDataControl;

```

```

GPIO_0(0) <= adcDat_sig;
GPIO_0(1) <= adcLRCK_sig;
GPIO_0(2) <= dacLRCK_sig;
GPIO_0(3) <= dacDat_sig;
GPIO_0(6) <= clock18MHz;
GPIO_0(7) <= bck_sig;

```

```

GPIO_0(4) <= reset;
GPIO_0(5) <= codecResetn;

```

```

LEDG(0) <= reset;

```

```

end toplevel;

```

audioCodecController.vhd

```

--4/14/11

```

```

--Controller for the Wolfson audio codec
--Uses I2C to initialize and send data to the codec
--Data is stored in a 24x10 bit ROM component.

```

```

library ieee;
use ieee.std_logic_1164.all;

```

```

entity audioCodecController is

```

```

    port(
        clock50MHz,reset: in std_logic;
        I2C_SCLK_Internal: out std_logic;
        --must be inout to allow FPGA to read the ack bit
        I2C_SDAT_Internal: out std_logic;
        SDAT_Control: out std_logic;
        --for testing
        clock50KHz_Out: out std_logic
    );

```

```

end audioCodecController;

```

```

architecture behavior of audioCodecController is

```

```

    --50KHz SCLK
    component clock50KHz is
        port(
            inClock,reset: in std_logic;
            outClock50KHz: out std_logic
        );
    end component;

```

```

--counts the number of data bits sent
component dataBitCounter is
    port(
        --active high count enable
        countEnable: in std_logic;
        --active high reset
        reset: in std_logic;
        clock: in std_logic;
        currentBitCount: out integer;
        currentWordCount: out integer
    );
end component;

--ROM storing codec initialization data.
--10 words, 24 bits each
component ROMcontroller is
    port(
        --asynch active-high reset
        reset: in std_logic;
        increment: in std_logic;
        clock50KHz: in std_logic;
        clock50MHz: in std_logic;
        ROMword: out std_logic_vector(23 downto 0)
    );
end component;

--50KHz clock used for SCLK
signal clock50KHz_Internal: std_logic;

--internal signals
signal SDAT_Temp,SCLK_Temp: std_logic;

--starts/stops the data bit counter
signal bitCountEnable: std_logic;

--start incrementing the ROM each clock cycle
signal incrementROM: std_logic;

--the 24 bits of data to be sent
signal ROM_data_vector_24: std_logic_vector(23 downto 0);

--track bit in current set of data (0 -> 23)
signal currentDataBit: integer;

--track current 24-bit word in ROM
signal currentDataWord: integer;

--each state places one bit on the SDAT wire
type I2CState_type is (resetState, startCondition, sendData, acknowledge, prepForStop, stopCondition);
signal I2C_state: I2CState_type;

```

begin

```
clock50KHzInstance: clock50KHz port map(clock50MHz,reset,clock50KHz_Internal);
dataBitCounterInstance: dataBitCounter port map(bitCountEnable, reset, clock50KHz_Internal, currentDataBit,
currentDataWord);
ROMcontrollerInstance: ROMcontroller port map(reset, incrementROM, clock50KHz_Internal, clock50MHz,
ROM_data_vector_24);
```

```
--FSM that sends start condition, address, write bit = 0,
--then waits for ack from the codec
```

```
process(clock50KHz_Internal,reset)
```

```
begin
```

```
--asynchronous active-high reset
```

```
if reset = '0' then
```

```
if rising_edge(clock50KHz_Internal) then
```

```
case I2C_state is
```

```
when resetState =>
```

```
--place both wires high to prepare for the start condition
```

```
SDAT_Temp <= '1';
```

```
SCLK_Temp <= '1';
```

```
I2C_state <= startCondition;
```

```
incrementROM <= '0';
```

```
when startCondition =>
```

```
--pull the SDAT line low -> the start condition
```

```
SDAT_Temp <= '0';
```

```
I2C_state <= sendData;
```

```
--start counting data bits on the next clock cycle
```

```
bitCountEnable <= '1';
```

```
when sendData =>
```

```
--release the clock
```

```
SCLK_Temp <= '0';
```

```
SDAT_Control <= '1';
```

```
--send the next data bit
```

```
SDAT_Temp <= ROM_data_vector_24(currentDataBit);
```

```
--is it time for the ack bit?
```

```
if (currentDataBit = 16) or (currentDataBit = 8) or (currentDataBit = 0)
```

```
then
```

```
I2C_state <= acknowledge;
```

```
bitCountEnable <= '0';
```

```
else
```

```
I2C_state <= sendData;
```

```
end if;
```

```
when acknowledge =>
```

```
--To allow the codec to pull SDAT low, SDAT must be set to Z
```

```
SDAT_Control <= '0';
```

```
--if all 24 bits sent, end the transmission
```

```
if currentDataBit = 23 then
```

```
I2C_state <= prepForStop;
```

```
else
```

```
I2C_state <= sendData;
```

```

        bitCountEnable <= '1';
    end if;
    when prepForStop =>
        --take control of SDAT line again
        SDAT_Control <= '1';
        --pull SCLK high, and set SDAT low to prep for stop condition
        SCLK_Temp <= '1';
        SDAT_Temp <= '0';
        I2C_state <= stopCondition;
    when stopCondition =>
        --keep SCLK high, and pull SDAT high as stop condition
        SDAT_TEMP <= '1';
        --more data words to send?
        --Note: currentDataWord = # of words already sent at this point
        if currentDataWord < 10 then
            incrementROM <= '1';
            I2C_state <= resetState;
        else
            incrementROM <= '0';
        end if;
    end case;
end if;
else
    SDAT_Temp <= '1';
    SCLK_Temp <= '1';
    SDAT_Control <= '1';
    bitCountEnable <= '0';
    incrementROM <= '0';
    I2C_state <= resetState;
end if;
end process;

I2C_SDAT_Internal <= SDAT_Temp;
--use the 50KHz clock to drive the state machine, and the (not 50KHz) clock to drive the
--codec. The Half-period delay allows the SDAT data to stabilize on the line before
--being read by the codec
I2C_SCLK_Internal <= SCLK_Temp or (not clock50KHz_Internal);

--for testing purposes
clock50KHz_Out <= clock50KHz_Internal;

end behavior;

```

delayCounter.vhd

```

--delayCounter
--after reset clears, waits for 40ms, then asserts a reset signal at the output
--input is the 50MHz clock from the clockBuffer
--40ms(50MHz) = 2x10^6 counts
--integer type can hold -(2^31-1) -> (2^31-1)

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

```

```

entity delayCounter is
    port(
        clock,reset: in std_logic;
        --active high reset
        resetAdc: out std_logic
    );
end delayCounter;

```

```

architecture behavior of delayCounter is

```

```

    --count up to 2x10^6
    signal count: integer range 0 to 1999999;
    --active-high output signal
    signal output: std_logic;

```

```

begin

```

```

    process(clock,reset,output)
    begin

```

```

        --asynchronous active-high reset
        --stop counting after setting the output
        if (reset = '0') then

```

```

            --synchronous count

```

```

            if rising_edge(clock) then

```

```

                if output = '0' then

```

```

                    if count = 1999999 then --1999 for 40us

```

```

                        count <= 0;

```

```

                        --count has reached 2x10^6 (40ms), assert resetAdc

```

```

                        output <= '1';

```

```

                    else

```

```

                        count <= count + 1;

```

```

                    end if;

```

```

                end if;

```

```

            end if;

```

```

        else

```

```

            --in reset

```

```

            count <= 0;

```

```

            --resetAdc is not asserted

```

```

            output <= '0';

```

```

        end if;

```

```

    end process;

```

```

    --assign output signal
    resetAdc <= output;

```

```

end behavior;

```

bclk_counter.vhd

```
--5/10/11
--bitCount counter for audio codec. Input is 18MHz clock (master clock) from PLL.
--1 period is 12 counts of the master clock => flip the output every 6 counts
```

```
library ieee;
use ieee.std_logic_1164.all;

entity bclk_counter is
    port(
        --active high reset
        reset: in std_logic;
        mclk: in std_logic;
        bclk: out std_logic
    );
end bclk_counter;
```

```
architecture behavior of bclk_counter is
```

```
    signal count: integer range 0 to 5 := 0;
    signal output: std_logic := '0';
```

```
begin
```

```
    process(reset, mclk)
    begin
        if reset = '1' then
            count <= 0;
        elsif rising_edge(mclk) then
            if count < 5 then
                count <= count + 1;
            else
                output <= not output;
                count <= 0;
            end if;
        end if;
    end process;
```

```
    bclk <= output;
```

```
end behavior;
```

LRchannelCounter.vhd

```
--5/10/11
--left channel, right channel counter for audio codec
--the state changes after 16 falling-edge cycles of the bitClock
```

```
library ieee;
use ieee.std_logic_1164.all;
```

```

entity LRchannelCounter is
    port(
        --active high reset
        reset: in std_logic;
        bclk: in std_logic;
        --left = '1', right = '0'
        LRchannel: out std_logic
    );
end LRchannelCounter;

```

architecture behavior of LRchannelCounter is

```

    signal count: integer range 0 to 15 := 0;
    signal output: std_logic := '1';

```

begin

```

    process(reset, bclk)
    begin
        if reset = '1' then
            output <= '1';
            count <= 0;
        elsif falling_edge(bclk) then
            if count < 15 then
                count <= count + 1;
            else
                output <= not output;
                count <= 0;
            end if;
        end if;
    end process;

    LRchannel <= output;

```

end behavior;

AdcDacController.vhd

```
--5/9/11
```

```
--Creates the ADC and DAC signals needed by the audio codec.
```

```
--inputs are reset from the delay buffer, 50MHz clock from PLL
```

```
library ieee;
```

```
use ieee.std_logic_1164.all;
```

```
entity AdcDacController is
```

```

    port(
        --reset signal starts '0', then goes to '1' after 40 ms => active-low
        resetn: in std_logic;
        --from 50MHz PLL at toplevel

```

```

        clock18MHz_in: in std_logic;
        --line-in on the DE1
        adcData: in std_logic;
        --line-out on the DE1
        dacData: out std_logic;
        bitClock: out std_logic;
        dacLRSelect: out std_logic;
        adcLRSelect: out std_logic;
        --neuron model signals
        neuronVin: in std_logic_vector(15 downto 0);
        neuronUin: in std_logic_vector(15 downto 0);
        --selects loopback or neuron signals using switches
        outputSelect: in std_logic_vector(1 downto 0)
    );
end AdcDacController;

```

architecture behavior of AdcDacController is

```

--bitCount generator. Changes every 12 counts of the master clock (18MHz)
component bclk_counter is

```

```

    port(
        --active high reset
        reset: in std_logic;
        mclk: in std_logic;
        bclk: out std_logic
    );
end component;

```

```

--generates left/right channel signal
component LRchannelCounter is

```

```

    port(
        --active high reset
        reset: in std_logic;
        bclk: in std_logic;
        --left = '1', right = '0'
        LRchannel: out std_logic
    );
end component;

```

```

--active-high reset
signal reset: std_logic;

```

```

--bit clock
signal bitClock_sig: std_logic;

```

```

--left/right channel control signal
signal LRchannel_sig: std_logic;

```

```

--counts the bit of neuron data to be sent
signal bitCounter: integer range 15 downto 0 := 15;

```

begin

```
--turns active-low reset into active-high  
reset <= not resetn;
```

```
bclk_counterMap: bclk_counter port map(reset, clock18MHz_in, bitClock_sig);  
LRchannelCounterMap: LRchannelCounter port map(reset, bitClock_sig, LRchannel_sig);
```

```
--output signals  
bitClock <= bitClock_sig;  
dacLRSelect <= LRchannel_sig;  
adcLRSelect <= LRchannel_sig;
```

```
--count out the neuron model bits  
process(bitClock_sig, bitCounter)  
begin  
    if rising_edge(bitClock_sig) then  
        if bitCounter > 0 then  
            bitCounter <= bitCounter - 1;  
        else  
            bitCounter <= 15;  
        end if;  
    end if;  
end process;
```

```
--select loopback test or neuron model DAC output  
process(neuronVin, neuronUin, adcData, outputSelect, bitCounter)  
begin  
    if outputSelect = "01" then  
        dacData <= neuronVin(bitCounter);  
    elsif outputSelect = "10" then  
        dacData <= neuronUin(bitCounter);  
    else  
        dacData <= adcData;  
    end if;  
end process;
```

end behavior;

neuronModel.vhd

--5/17/11

--Neuron model component

--sends data to the adcDacController.

--Model output is produced by the audio codec DAC, and can be viewed on the oscilloscope

--by connecting to the DE1 audio line-out

LIBRARY ieee;

USE ieee.std_logic_1164.all;

use ieee.std_logic_signed.all;

```

entity neuronModel is
  port(
    frameClock: in std_logic;
    resetn: in std_logic;
    vOut: out std_logic_vector(17 downto 0);
    uOut: out std_logic_vector(17 downto 0)
  );
end neuronModel;

```

architecture behavior of neuronModel is

```

  component signedMultiplier is
    port(
      multiplier: in std_logic_vector(17 downto 0);
      multiplicand: in std_logic_vector(17 downto 0);
      product: out std_logic_vector(17 downto 0)
    );
  end component;

  --state variables
  signal v, vNew, vSquared, vdt, u, uNew, uReset, udt, vb, vbMinusU: std_logic_vector(17 downto 0) :=
'0'&'0'&'X"0000";
  --parameters and constants
  signal a, b, c, d, p, c14, l: std_logic_vector(17 downto 0) := '0'&'0'&'X"0000";
  --prescalar of 4096
  signal count: integer range 0 to 8 := 0;

```

begin

```

  --this is the synchronous state variable update,
  --at 12 KHz (8 counts of the 96 KHz frame clock)
  process(frameClock, resetn)
  begin
    --initialize constants
    a <= '0'&'0'&'X"051E"; --18'h0051E
    b <= '0'&'0'&'X"3333"; --18'h03333 = 0.2
    c <= '1'&'1'&'X"8000"; -- 18'h260000= -0.625
    d <= '0'&'0'&'X"051E"; -- 18'h01478 = 0.08
    p <= '0'&'0'&'X"4CCC"; -- 18'h16666; = 1.4
    c14 <= '0'&'1'&'X"6666"; -- 18'h16666; = 1.4
    l <= '0'&'0'&'X"2666"; -- 18'h02666 = 0.15

    if resetn = '0' then
      count <= 0;
      -- initial state
      v <= '1'&'1'&'X"4CCD"; -- 0x34CCD = -0.7
      u <= '1'&'1'&'X"CCCD"; -- 0x3CCCD = -0.2
    else
      if rising_edge(frameClock) then
        if count = 8 then
          count <= 0;

```

```

else
    count <= count + 1;
end if;

if count = 0 then
    if v > p then
        v <= c;
        u <= uReset;
    else
        v <= vNew;
        u <= uNew;
    end if;
end if;
end if;
end if;
end process;

--Euler's method
--dt = 1/16
--Thus, time scale is actualy (12e3/16)^-1 = 1.3 ms
--v1(n+1) = v1(n) + (v1(n)^2) + 5/4*v1(n) + 1.40/4 - u1(n)/4 + 1/4/4
squareV: signedMultiplier port map(v, v, vSquared);
vdt <= vSquared + (v + ((v(17)&v(17)&v(17 downto 2)))) + (c14(17)&c14(17)&c14(17 downto 2)) -
    (u(17)&u(17)&u(17 downto 2)) + (l(17)&l(17)&l(17 downto 2));
vNew <= v + (vdt(17)&vdt(17)&vdt(17 downto 2));

--u1(n+1) = u1 + dt*a*(b*v1(n) - u1(n));
vTimesb: signedMultiplier port map(v, b, vb);
vbMinusU <= vb - u;
axVbMinusU: signedMultiplier port map(a, vbMinusU, udt);
uNew <= u + (udt(17)&udt(17)&udt(17 downto 2));
uReset <= u + d;

--state outputs
vOut <= v;
uOut <= u;

end behavior;

```

signedMultiplier.vhd

--5/17/11

--Multiplies two 18-bit signed numbers,
--returns the 18 most significant bits

```

LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_signed.all;

```

entity signedMultiplier is

```

    port(
        multiplier: in std_logic_vector(17 downto 0);
        multiplicand: in std_logic_vector(17 downto 0);
        product: out std_logic_vector(17 downto 0)
    );
end signedMultiplier;

```

architecture behavior of signedMultiplier is

```

    signal product_vector: std_logic_vector(35 downto 0);

begin

    product_vector <= multiplier * multiplicand;
    --sign bit is bit 35. The bottom 32 bits are the least significant decimal bits and are ignored. Bits 34 and 33 are
the overflow bits
    product <= product_vector(35) & product_vector(32 downto 16);

end behavior;

```

ROMcontroller.vhd

```

--5/3/11
--using input from the audio codec, selects and returns a 24-bit word from the rom
--input is an increment signal, which causes the address of the ROM to increment by one,
--from 0 to 10

```

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

```

```

entity ROMcontroller is
    port(
        --asynch active-high reset
        reset: in std_logic;
        increment: in std_logic;
        clock50KHz: in std_logic;
        clock50MHz: in std_logic;
        ROMword: out std_logic_vector(23 downto 0)
    );
end ROMcontroller;

```

architecture behavior of ROMcontroller is

```

--ROM 1-port memory module from MegaIP Wizard
component codecROM IS
    PORT
    (
        address      : IN STD_LOGIC_VECTOR (4 DOWNTO 0);
        clock        : IN STD_LOGIC := '1';
        q            : OUT STD_LOGIC_VECTOR (23 DOWNTO 0)
    )

```

```

    );
END component;

--address vector sent to the ROM component
signal address_vector_5: std_logic_vector(4 downto 0);
signal address_integer: integer range 0 to 9 := 0;
--output data vector from the ROM component
signal data_vector_24: std_logic_vector(23 downto 0);

begin

codecROMInstance: codecROM port map(address_vector_5, clock50MHz, data_vector_24);

process(clock50KHz, reset)
begin
    if reset = '0' then
        if rising_edge(clock50KHz) then
            if increment = '1' then
                if address_integer < 9 then
                    address_integer <= address_integer + 1;
                else
                    address_integer <= 0;
                end if;
            end if;
        end if;
    else
        address_integer <= 0;
    end if;
end process;

--convert address integer into address vector
address_vector_5 <= std_logic_vector(to_unsigned(address_integer, 5));

ROMword <= data_vector_24;

end behavior;

```

dataBitCounter.vhd

```

--5/2/11
--synchronous counter using the 50khz clock from the audioCodecController
--counts the dataBits (0 to 23) sent to the audio codec

library ieee;
use ieee.std_logic_1164.all;

entity dataBitCounter is
    port(
        --active high count enable
        countEnable: in std_logic;

```

```

        --active high reset
        reset: in std_logic;
        clock: in std_logic;
        currentBitCount: out integer;
        currentWordCount: out integer
    );
end dataBitCounter;

```

architecture behavior of dataBitCounter is

```

    --output
    signal countBit: integer range 0 to 23 := 23;
    signal countWord: integer range 0 to 10 := 0;

begin

    --starts counting when rest is cleared and enable is
    process(clock, reset, countEnable)
    begin
        if reset = '0' then
            if rising_edge(clock) then
                if countEnable = '1' then
                    if countBit > 0 then
                        countBit <= countBit - 1;
                    else
                        countBit <= 23;
                        if countWord < 10 then
                            countWord <= countWord + 1;
                        else
                            countWord <= 0;
                        end if;
                    end if;
                end if;
            end if;
        else
            countBit <= 23;
            countWord <= 0;
        end if;
    end process;

    currentBitCount <= countBit;
    currentWordCount <= countWord;

```

end behavior;

clock50KHz.vhd

```

--50KHz clock for I2C module
--input is 50MHz clock from PLL clockBuffer
--output is 50KHz clock
--50MHz/1000 = 50KHz => count to 500, then not the output

```

```

library ieee;
use ieee.std_logic_1164.all;

entity clock50KHz is
    port(
        inClock,reset: in std_logic;
        outClock50KHz: out std_logic
    );
end clock50KHz;

architecture behavior of clock50KHz is

    --count to half the period (500)
    signal count: integer range 0 to 499;
    --output 50KHz clock signal
    signal output: std_logic;

begin

    process(inClock,reset)
    begin
        --asynchronous active-high reset
        if reset = '0' then
            --synchronous count
            if rising_edge(inClock) then
                if count = 499 then
                    count <= 0;
                    --count has reached 500 (half-period)
                    output <= not output;
                else
                    count <= count + 1;
                end if;
            end if;
        else
            --in reset
            count <= 0;
            output <= '0';
        end if;
    end process;

    --assign output signal
    outClock50KHz <= output;
end behavior;

```