Kiana Peterson and Nattapon Oonlmaom
EE 371
February 24, 2023
Lab5: Digital Signal Processing

Procedure
The purpose and objective of this lab was to use CODEC (audio coder/decoder) to play sounds on the DE1-SoC board from both an audio file and ROM memory.

Task 1:
In Task 1, we edited part1.v from the starter code to play audio into the DE1-SoC from the given audio file.

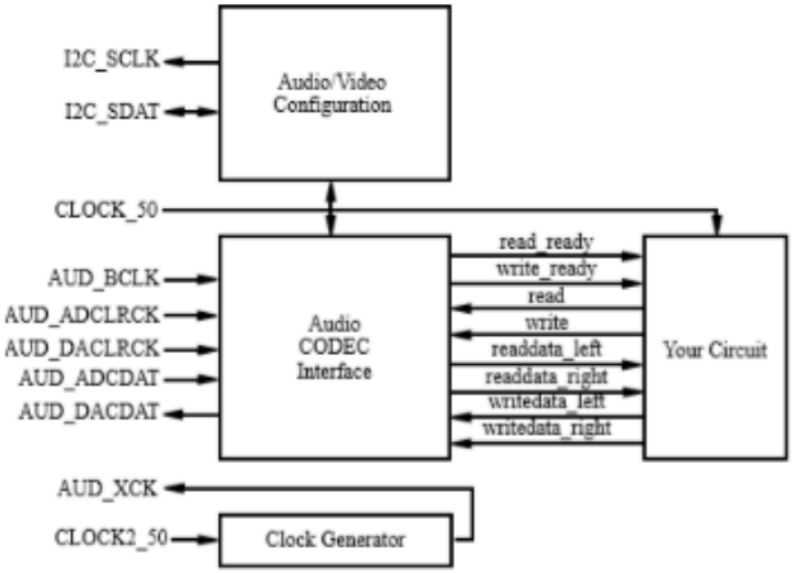

Figure 1.1 Block Diagram for Task1

Task 2:
In Task 2, we created a system that would generate a sound from frequencies stored in a ROM (generated via a mif file called tone.mif). This system worked similarly to task 1, except it used a counter corresponding to addresses in the ROM and looped through to play the sound. There were 48000 values stored in the ROM, which requires 16 bits. Instead of looping through 48000 frequencies, we looped through 185 because that was essentially the period of the tone. This allowed us to use an 8-bit counter rather than a 16-bit counter. The frequency is then assigned to writedata_left and writedata_right and sent to the CODEC. In the final part of task 2, the top level module (part2.sv) combines the already written task 2 with task 1 by switching where the sound is being outputted from. If SW9 is low, the provided audio file will play, if it is high, the sound generated from the ROM will play.

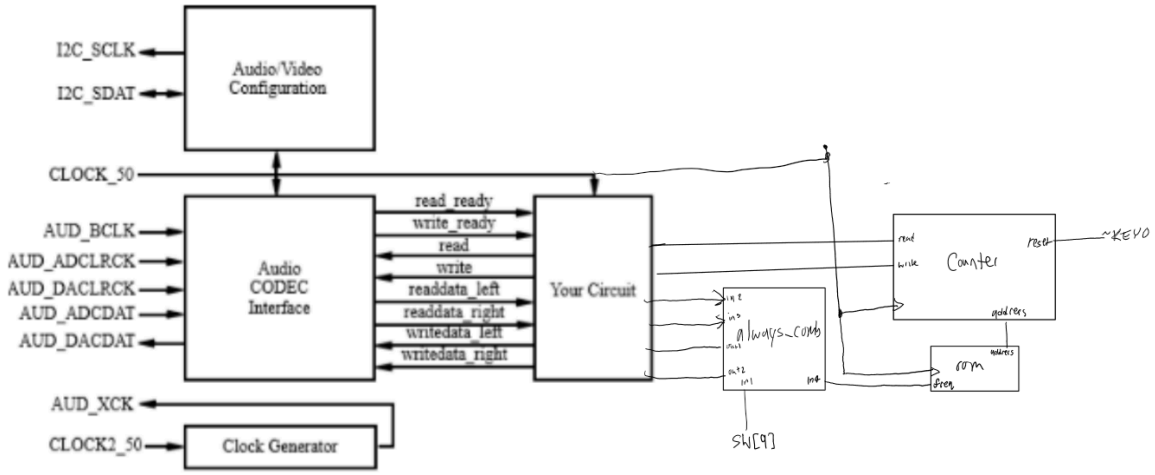

Figure 2.1 Block Diagram for Task2

Results

Task 1:

No waveform needed/tested. The audio is played and recorded on LabsLand. The module part1.v functioned correctly as we intended it to!

Task 2:

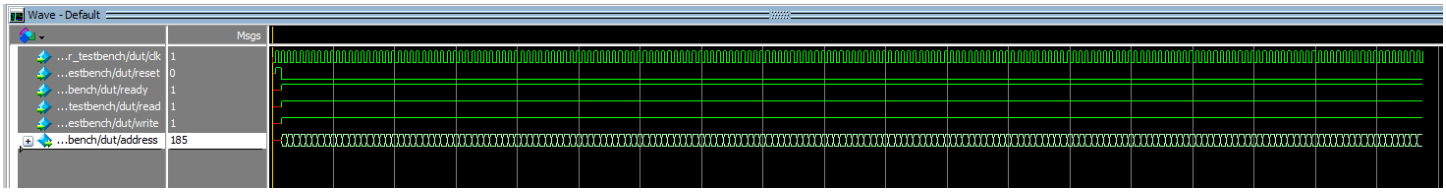

Figure 3.1 counter.sv Waveform

The counter module works as expected. It cycles from 0 to 185 by 1 and repeats that cycle. See Figure 3.2 below for a zoomed in waveform.

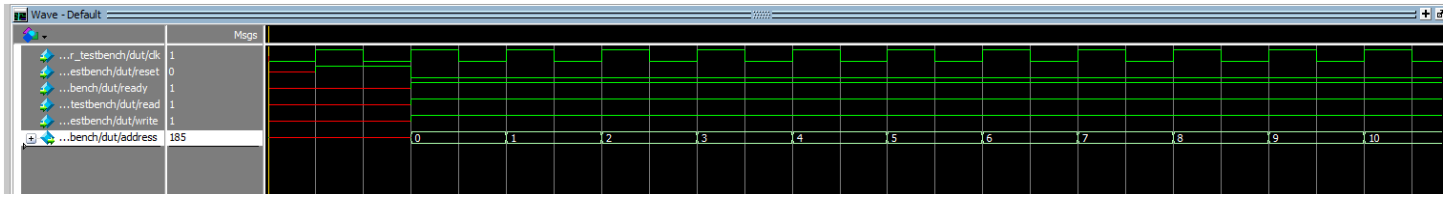

Figure 3.2 counter.sv Waveform (Zoomed)

Final Product

Overall, the product of this project was an implementation of audio being played/created via two different sources. The first source being an audio file, the second source being ROM memory. The ROM memory is

filled with frequencies created from a mif file. The mif file was a product of running a python script that creates mif files full of frequencies corresponding to a specific note and length. The final product was shown in a demo by switching from one source to another by using switch 9.

Appendix: SystemVerilog Code

```
// megafunction wizard: %RAM: 1-PORT%
// GENERATION: STANDARD
// VERSION: WM1.0
// MODULE: altsyncram

// ============================================================
// File Name: rom.v
// Megafunction Name(s):
//          altsyncram
//
// Simulation Library Files(s):
//          altera_mf
// ============================================================
// ************************************************************
// THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
//
// 17.0.0 Build 595 04/25/2017 SJ Lite Edition
// ************************************************************


//Copyright (C) 2017  Intel Corporation. All rights reserved.
//Your use of Intel Corporation's design tools, logic functions
//and other software and tools, and its AMPP partner logic
//functions, and any output files from any of the foregoing
//(including device programming or simulation files), and any
//associated documentation or information are expressly subject
//to the terms and conditions of the Intel Program License
//Subscription Agreement, the Intel Quartus Prime License Agreement,
//the Intel MegaCore Function License Agreement, or other
//applicable license agreement, including, without limitation,
//that your use is for the sole purpose of programming logic
//devices manufactured by Intel and sold by Intel or its
//authorized distributors.  Please refer to the applicable
//agreement for further details.


// synopsys translate_off
`timescale 1 ps / 1 ps
// synopsys translate_on
module rom (
    address,
    clock,
    data,
    wren,
    q);

    input [7:0]  address;
    input   clock;
    input [23:0]  data;
    input   wren;
    output  [23:0]  q;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_off
`endif
    tri1    clock;
`ifndef ALTERA_RESERVED_QIS
// synopsys translate_on
`endif

    wire [23:0] sub_wire0;
    wire [23:0] q = sub_wire0[23:0];

    altsyncram  altsyncram_component (
                .address_a (address),
                .address_a (address),
                .clock0 (clock),
                .data_a (data),
                .wren_a (wren),
                .q_a (sub_wire0),
                .aclr0 (1'b0),
                .aclr1 (1'b0),
                .address_b (1'b1),
                .addressstall_a (1'b0),
                .addressstall_b (1'b0),
                .byteena_a (1'b1),
                .byteena_b (1'b1),
                .clock1 (1'b1),
                .clocken0 (1'b1),
                .clocken1 (1'b1),
                .clocken2 (1'b1),
                .clocken3 (1'b1),
                .data_b (1'b1),
                .eccstatus (),
                .q_b (),
                .rden_a (1'b1),
                .rden_b (1'b1),
                .wren_b (1'b0));
    defparam
        altsyncram_component.clock_enable_input_a = "BYPASS",
        altsyncram_component.clock_enable_output_a = "BYPASS",
        altsyncram_component.init_file = "notes.mif",
        altsyncram_component.intended_device_family = "Cyclone V",
        altsyncram_component.lpm_hint = "ENABLE_RUNTIME_MOD=NO",
        altsyncram_component.lpm_type = "altsyncram",
        altsyncram_component.numwords_a = 256,
        altsyncram_component.operation_mode = "SINGLE_PORT",
        altsyncram_component.outdata_aclr_a = "NONE",
        altsyncram_component.outdata_reg_a = "CLOCK0",
        altsyncram_component.power_up_uninitialized = "FALSE",
        altsyncram_component.ram_block_type = "M10K",
        altsyncram_component.read_during_write_mode_port_a = "NEW_DATA_NO_NBE_READ",
        altsyncram_component.widthad_a = 8,
        altsyncram_component.width_a = 24,
        altsyncram_component.width_byteena_a = 1;


endmodule

// ============================================================
// CNX file retrieval info
// ============================================================
// Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
// Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
// Retrieval info: PRIVATE: AclrByte NUMERIC "0"
// Retrieval info: PRIVATE: AclrData NUMERIC "0"
// Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
// Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
// Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
// Retrieval info: PRIVATE: Clken NUMERIC "0"
// Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
// Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
// Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
// Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
// Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
// Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
```

Figure 4 rom.v

Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
0	0	287231	574125	860346	1145557	1429426	1711618	1991802	.....
8	2269651	2544838	2817040	3085939	3351219	3612568	3869681	4122255	.....
16	4369995	4612611	4849816	5081334	5306893	5526228	5739082	5945206	.....
24	6144357	6336302	6520817	6697684	6866696	7027655	7180373	7324669	.....
32	7460376	7587333	7705392	7814415	7914273	8004849	8086038	8157744	.....
40	8219882	8272381	8315178	8348224	8371479	8384916	8388520	8382286	.....
48	8366221	8340345	8304688	8259291	8204208	8139503	8065253	7981544	.....
56	7888475	7786155	7674703	7554250	7424938	7286919	7140354	6985414	.....
64	6822283	6651150	6472218	6285695	6091800	5890761	5682814	5468202	.....
72	5247177	5019999	4786933	4548253	4304240	4055178	3801361	3543085	.....
80	3280655	3014377	2744564	2471532	2195601	1917096	1636342	1353670	.....
88	1069409	783895	497461	210444	16700396	16413223	16126476	15840492	.....
96	15555607	15272154	14990467	14710875	14433706	14159286	13887935	13619974	.....
104	13355715	13095468	12839540	12588229	12341831	12100635	11864923	11634972	.....
112	11411052	11193425	10982347	10778064	10580817	10390837	10208347	10033560	.....
120	9866683	9707909	9557426	9415411	9282028	9157436	9041780	8935196	.....
128	8837809	8749733	8671071	8601916	8542348	8492438	8452244	8421814	.....
136	8401182	8390373	8389400	8398264	8416954	8445449	8483716	8531708	.....
144	8589371	8656635	8733424	8819646	8915200	9019974	9133846	9256682	.....
152	9388338	9528659	9677480	9834628	9999919	10173157	10354140	10542656	.....
160	10738484	10941393	11151147	11367498	11590194	11818974	12053568	12293701	.....
168	12539093	12789455	13044494	13303910	13567400	13834654	14105359	14379198	.....
176	14655848	14934987	15216286	15499416	15784044	16069837	16356460	16643576	.....
184	153633	440725	727301	1013024	1297559	1580572	1861731	2140707	.....
192	2417173	2690804	2961279	3228281	3491497	3750618	4005341	4255367	.....
200	4500402	4740159	4974357	5202722	5424984	5640885	5850170	6052594	.....
208	6247921	6435919	6616370	6789062	6953791	7110366	7258601	7398324	.....

Figure 5 notes.mif

```
1  // Nattapon Oonlamom and Kiana Peterson
2  // 02/25/2023
3  // Lab 5: Digital Signal Processing, Task 2
4
5  // This module counts from 0 to the MAX number set
6  // Since output is 8 bits, the MAX is 184 in this case
7  // 185 was used intead of 48000 (the actual size of the
8  // file given because of the repetitive notes)
9  // If MAX = 48000, the width = 16
10
11 // Overall inputs and outputs for the counter module listed below:
12 // Inputs: 1-bit read, write, reset, clk
13 // Outputs: (in this case) 8-bit address
14 // Parameter: MAX, width for easy changes
15 module counter #(parameter MAX = 184, width = 8)
16                 (read, write, reset, clk, address);
17    input  logic                clk, reset;
18    input  logic                read, write;
19    output logic [width - 1:0] address;
20
21    // sequential logic
22    always_ff @(posedge clk) begin
23       if (reset || (address == MAX)) begin
24          address <= 16'b0;
25       end
26       else if (read && write) begin
27           address <= address + 1'b1;
28       end
29    end
30 endmodule
31
32 // Testbench for counter module to test every possible combination of inputs
33 // to see if the module is working correctly
34 module counter_testbench();
35
36    // logic to stimulate
37    logic clk, reset;
38    logic read, write;
39    logic [23:0] address;
40    logic clock;
41    logic q;
42
43    // intantiate counter testbench
44    counter dut(.read, .write, .reset, .clk, .address);
45    rom dut2(.address, .clock, .q);
46
47    // Set up the clock.
48    parameter CLOCK_PERIOD = 100;
49    initial begin
50       clk <= 0;
51       forever #(CLOCK_PERIOD/2) clk <= ~clk;
52    end // initial
53
54    initial begin
55                                                          @(posedge clk); // Neither reset or w have values
56       reset <= 1;                                        @(posedge clk); // At next leading edge, reset = 1
57       reset <= 0; read <= 1; write <= 1;                 @(posedge clk); // At next leading edge, reset = 0
58                                              repeat (184) @(posedge clk);
59       $stop; // End the simulation.
60    end
61 endmodule
```

Figure 6 counter.sv

```
// This is the top-level module for the audio coder/decoder (CODEC)
// This main module record and outputs the given mp3 with SW[9]
// is not pressed, but outputs the tones from the mif file when
// SW[9] is pressed.

/* Overall inputs and outputs for the counter module listed below:
 * Inputs: CLOCK_50, CLOCK2_50
 *         1 -bit KEY (active high)
 *         10-bit SW
 *         FPGA_I2C_SDAT
 *         AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK
 *         AUD_ADCDAT
 * Outputs:
 *         FPGA_I2C_SCLK
 *         AUD_XCK
 *         AUD_DACDAT
 */
module part2 (CLOCK_50, CLOCK2_50, KEY, SW, FPGA_I2C_SCLK, FPGA_I2C_SDAT, AUD_XCK,
              AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT);

   input logic CLOCK_50, CLOCK2_50; // 50mHz Clock
   input logic [0:0] KEY;
   input logic [9:0] SW;

   // I2C Audio/Video config interface
   output FPGA_I2C_SCLK;
   inout  FPGA_I2C_SDAT;

   // Audio CODEC
   output AUD_XCK;
   input  AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK;
   input  AUD_ADCDAT;
   output AUD_DACDAT;

   // Local wires.
   wire read_ready, write_ready, read, write;
   wire [23:0] readdata_left, readdata_right;
   wire [23:0] writedata_left, writedata_right;
   wire reset = ~KEY[0];

    // logic to use
   logic [7:0] address;    // change accordingly to the param of counter.sv
   logic [23:0] freq;

   assign read  = read_ready & write_ready;
   assign write = write_ready & read_ready;

   // instantiated the counter module to loop throught the rom addresses
   counter addrCount (.read, .write, .reset, .clk(CLOCK_50), .address(address));

   // instantiated the rom stored memory to be read and write
   rom sample (.address(address), .clock(CLOCK_50), .q(freq));

   // counter only when SW[9] == 1
   always_comb begin
      if(SW[9] == 1) begin // Task2
         writedata_left  = freq;
         writedata_right = freq;
      end
      else begin  // Task1
         writedata_left  = readdata_left;
         writedata_right = readdata_right;
      end
   end

////////////////////////////////////////////////////////////////////////////////
// Audio CODEC interface.
//
// The interface consists of the following wires:
// read_ready, write_ready - CODEC ready for read/write operation
// readdata_left, readdata_right - left and right channel data from the CODEC
// read - send data from the CODEC (both channels)
// writedata_left, writedata_right - left and right channel data to the CODEC
// write - send data to the CODEC (both channels)
// AUD_* - should connect to top-level entity I/O of the same name.
//         These signals go directly to the Audio CODEC
// I2C_* - should connect to top-level entity I/O of the same name.
//         These signals go directly to the Audio/Video Config module
////////////////////////////////////////////////////////////////////////////////
   clock_generator my_clock_gen(
      // inputs
      CLOCK2_50,
      reset,

      // outputs
      AUD_XCK
   );

   audio_and_video_config cfg(
      // Inputs
      CLOCK_50,
      reset,

      // Bidirectionals
      FPGA_I2C_SDAT,
      FPGA_I2C_SCLK
   );

   audio_codec codec(
      // Inputs
      CLOCK_50,
      reset,

      read, write,
      writedata_left, writedata_right,

      AUD_ADCDAT,

      // Bidirectionals
      AUD_BCLK,
      AUD_ADCLRCK,
      AUD_DACLRCK,

      // Outputs
      read_ready, write_ready,
      readdata_left, readdata_right,
      AUD_DACDAT
   );

endmodule
```

Figure 7 part2.sv

```
// Nattapon Oonlamom and Kiana Peterson
// 02/25/2023
// Lab 5: Digital Signal Processing, Task 2

// This is the top-level module for the audio coder/decoder (CODEC)
// This main module outputs the given mp3

/* Overall inputs and outputs for the counter module listed below:
 * Inputs: CLOCK_50, CLOCK2_50
 *         1 -bit KEY (active high)
 *         FPGA_I2C_SDAT
 *         AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK
 *         AUD_ADCDAT
 * Outputs:
 *         FPGA_I2C_SCLK
 *         AUD_XCK
 *         AUD_DACDAT
 */
module part1 (CLOCK_50, CLOCK2_50, KEY, FPGA_I2C_SCLK, FPGA_I2C_SDAT, AUD_XCK,
              AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK, AUD_ADCDAT, AUD_DACDAT);

    input CLOCK_50, CLOCK2_50;
    input [0:0] KEY;
    // I2C Audio/Video config interface
    output FPGA_I2C_SCLK;
    inout FPGA_I2C_SDAT;
    // Audio CODEC
    output AUD_XCK;
    input AUD_DACLRCK, AUD_ADCLRCK, AUD_BCLK;
    input AUD_ADCDAT;
    output AUD_DACDAT;

    // Local Wires.
    wire read_ready, write_ready, read, write;
    wire [23:0] readdata_left, readdata_right;
    wire [23:0] writedata_left, writedata_right;
    wire reset = ~KEY[0];

    /////////////////////////////////
    // Your code goes here
    /////////////////////////////////
    assign writedata_left = readdata_left;
    assign writedata_right = readdata_right;
    assign read = read_ready & write_ready;
    assign write = read_ready & write_ready;


////////////////////////////////////////////////////////////////////////////////
// Audio CODEC interface.
//
// The interface consists of the following wires:
// read_ready, write_ready - CODEC ready for read/write operation
// readdata_left, readdata_right - left and right channel data from the CODEC
// read - send data from the CODEC (both channels)
// writedata_left, writedata_right - left and right channel data to the CODEC
// write - send data to the CODEC (both channels)
// AUD_* - should connect to top-level entity I/O of the same name.
//         These signals go directly to the Audio CODEC
// I2C_* - should connect to top-level entity I/O of the same name.
//         These signals go directly to the Audio/Video Config module
////////////////////////////////////////////////////////////////////////////////
    clock_generator my_clock_gen(
        // Inputs
        CLOCK2_50,
        reset,

        // outputs
        AUD_XCK
    );

    audio_and_video_config cfg(
        // Inputs
        CLOCK_50,
        reset,

        // Bidirectionals
        FPGA_I2C_SDAT,
        FPGA_I2C_SCLK
    );

    audio_codec codec(
        // Inputs
        CLOCK_50,
        reset,

        read, write,
        writedata_left, writedata_right,

        AUD_ADCDAT,

        // Bidirectionals
        AUD_BCLK,
        AUD_ADCLRCK,
        AUD_DACLRCK,

        // Outputs
        read_ready, write_ready,
        readdata_left, readdata_right,
        AUD_DACDAT
    );

endmodule
```

Figure 8 part1.v