

KVIC 336.007

16 b Delta-Sigma Audio DAC

© Harald Pretl
Institute for Integrated Circuits, Johannes Kepler University

v0.3 WS2022/23

Contents

1	Introduction	2
2	Description of the audio-DAC	2
3	Organization	2
4	Architecture	2
4.1	Host Interface	3
5	Tools	4
6	Background theory on delta-sigma modulators	5
7	Digital design	10
7.1	Quick-start guide and tutorial	11
8	Exemplary implementation and verification scripts	12
8.1	Matlab/Octave code for architecture development	12
8.2	Verilog code for delta-sigma modulator	13
8.3	Python3-driven Verilog test-bench	16
9	Optional work packages	20
9.1	Test modes	20
9.2	Wishbone interface	20
9.3	Higher-order modulator	20

Table 1: Specification and target performance

Specification	Value	Unit
Audio data input bit width (UINT)	16	bit
Audio input sample rate	8 to 48	kHz
Audio input interface	FIFO	—
Delta-sigma modulator	1st/2nd order error-FB	—
Modulator output	1	bit
OSR (bitstream clock / input sample rate)	64 to 256	—

1 Introduction

The target of the KV **Entwurf Integrierter Schaltungen** (336.007) in this semester is to design the digital part of an audio-DAC, to be implemented within one week (40 h). We are using open source IC design tools and the SkyWater SKY130 130 nm CMOS process (<https://skywater-pdk.readthedocs.io>). The target is to arrive at a complete design that could be submitted to IC fabrication! We also use important results from digital signal processing to arrive at a very efficient and simplified architecture.

2 Description of the audio-DAC

To simplify the design, we are targeting a delta-sigma DAC with a single-bit output. The delta-sigma DAC will be based on a first-order modulator structure¹ to simplify the design, and will include a FIFO towards a host system². This architecture allows for the implementation of large parts in an HDL (we will use **Verilog**). Despite this simplified architecture we will implement a high performance 16 bit DAC with $f_s = 8 \text{ kHz}$ to 48 kHz . A summary of the performance and target specification can be found in table 1.

3 Organization

Each student has to work on the design by herself/himself. At the end of the week, a final report about the work has to be submitted which will be used for grading (and automatically checked using TurnItIn).

To allow self-checking of the digital implementation, a verification framework will be provided to check the proper functionality.

4 Architecture

The overall architecture of the implemented audio-DAC can be seen in the block diagram in fig. 1. The interface signals to the DAC IP block are summarized in table 2.

¹Alternatively, a second-order modulator with better performance can be implemented if time allows.

²Optionally, if there is enough time left, then a WishBone bus interface can be implemented.

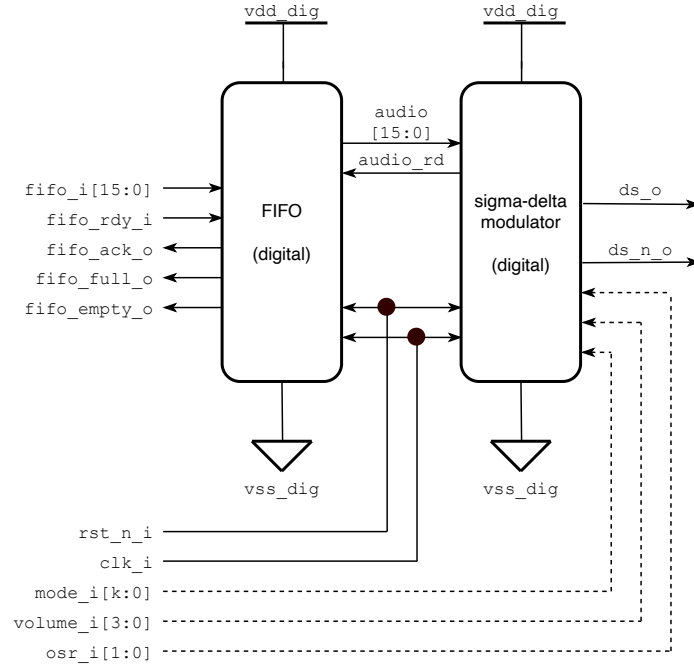


Figure 1: Audio-DAC high-level architecture.

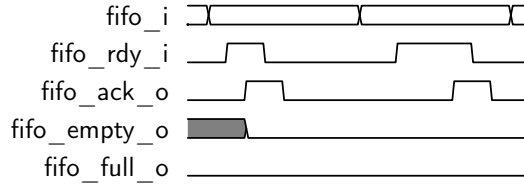


Figure 2: Host I/F timing.

4.1 Host Interface

To develop a flexible IP block an asynchronous FIFO-based host interface is specified. The detailed implementation (FIFO style, FIFO depth, etc.) can be freely chosen, but the FIFO interface shall work as described. To write into the FIFO:

1. The host asserts the new data word at `fifo_i[15:0]` (while `fifo_rdy_i=0` and `fifo_ack_o=0`)
2. The host signals the FIFO that data is ready with a transition of `fifo_rdy_i=0->1`
3. The host waits until the data is accepted by the FIFO by waiting for the ack signal being asserted (`fifo_ack_o=0->1`)
4. De-assert `fifo_rdy_i=1->0` (this is followed by `fifo_ack_o=1->0`)

This behaviour is illustrated in fig. 2.

Table 2: Interface description

Signal name	Description	Type
<code>fifo_i[15:0]</code>	Digital audio data input into FIFO, signed INT	dig-I
<code>fifo_rdy_i</code>	Digital audio data ready	dig-I
<code>fifo_ack_o</code>	Digital audio data accepted into FIFO	dig-O
<code>fifo_full_o</code>	Indicator that FIFO is full, no data is loaded into FIFO	dig-O
<code>fifo_empty_o</code>	Indicator that FIFO is empty	dig-O
<code>rst_n_i</code>	Reset (active low)	dig-I
<code>clk_i</code>	Audio clock ($\text{OSR} \times \text{audio input rate}$)	dig-I
<code>mode_i[k:0]</code>	Selection if first-order (<code>mode[0]=0</code>) or second-order modulation (<code>mode[0]=1</code>) (<i>implementation optional; mode[k:1] for other modes or test modes</i>)	dig-I
<code>volume_i[3:0]</code>	Volume control (<code>volume=15</code> maximum, <code>volume=0</code> turn output off) (<i>implementation optional</i>)	dig-I
<code>osr_i[1:0]</code>	Oversampling ratio (OSR), <code>osr=0</code> is 32, <code>osr=1</code> is 64, <code>osr=2</code> is 128, <code>osr=3</code> is 256 (<i>implementation optional</i>)	dig-I
<code>vdd_dig</code>	Digital block supply voltage (1.8 V)	VDD
<code>vss_dig</code>	Digital block ground	VSS

Data is only accepted by the FIFO if the FIFO is not full (indicated by `fifo_full_o=0`). Using `fifo_full_o` and `fifo_empty_o` an interrupt service routine (ISR) can be constructed:

1. `fifo_empty_o=0->1` triggers the ISR
2. The ISR writes data into the FIFO until it is full (by waiting for `fifo_full_o=0->1`)
3. Exit the ISR

This behavior is implemented in the Verilog verification test bench.

5 Tools

In this course, we will stick to open-source development tools. The architectural investigations will be done using **Octave** or **Python3**. The digital design will use **Icarus Verilog** for simulation, **gtkwave** for waveform viewing, and the implementation flow from **OpenLane**. All of these tools you can find on the Internet, and the IIC is providing a Docker-based solution that will be used during this course. The installation guideline of our software collection can be found at <https://github.com/iic-jku/iic-osic-tools>, please check the README.md.

In the repository at <https://github.com/iic-jku/iic-osic>, there is also an `example` folder, containing simple analog and digital designs to get you started.

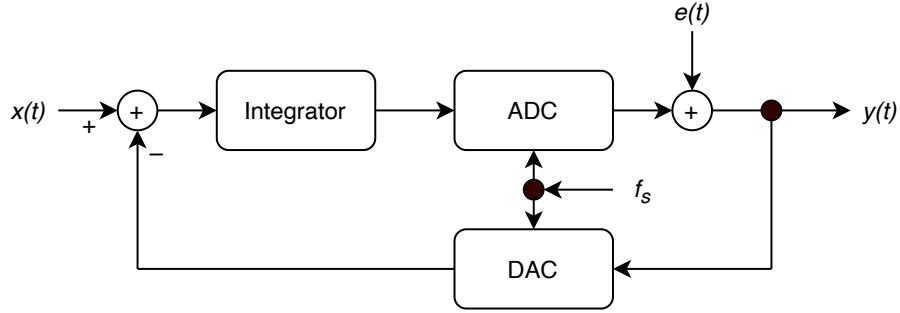


Figure 3: A simple analog-digital delta-sigma modulator architecture.

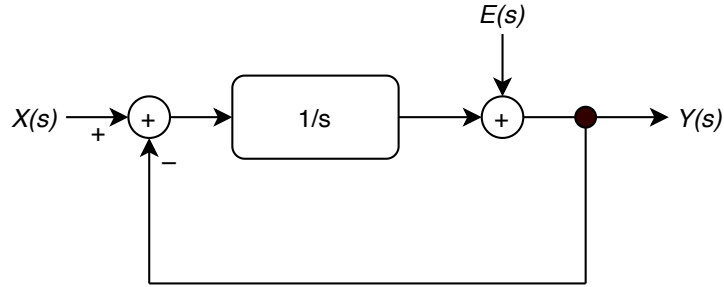


Figure 4: s -domain model of the simple delta-sigma modulator.

The idea of this lecture is that we learn together! It is planned that the course participants learn the required knowledge on-the-fly from the available online sources. Of course, there will be IIC staff in the room at all times, so that we can explore and learn together, and experiences can be shared. It is envisioned that this will be a fun and productive learning experience for everyone!

6 Background theory on delta-sigma modulators

A delta-sigma modulator can be designed according to the arrangement shown in fig. 3. This implements an ADC with a high effective resolution (higher than the number of physically implemented bits in the ADC and DAC) if the sampling frequency f_s is much higher than the maximum input frequency ($f_s \gg f_{\text{in,max}}$). The integrator in the feedback loop forces the output signal $y(t)$ to follow the input signal $x(t)$, at least for low frequencies, as the resulting error signal $x(t) - y(t)$ has to be zero at dc (otherwise the integrator output would not be stable). The quantization error introduced by the ADC is modeled by adding $e(t)$ to the output signal $y(t)$. The resulting model in the Laplace domain is shown in fig. 4.

The transfer function $H_X(s)$ from input to output is given by (we set $E(s) = 0$):

$$Y(s) = \frac{1}{s} [X(s) - Y(s)] \rightarrow Y(s) \left(1 + \frac{1}{s}\right) = \frac{1}{s} X(s) \quad (1)$$

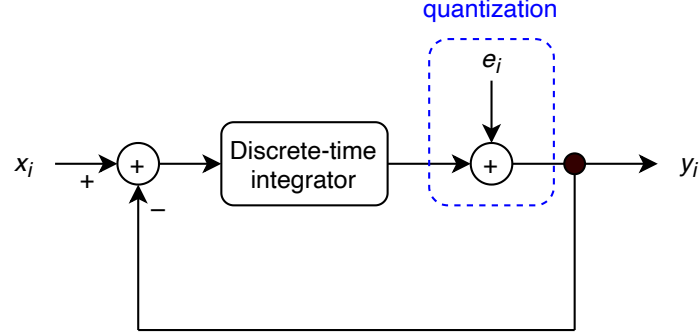


Figure 5: Discrete time version of the digital delta-sigma modulator.

Further manipulation of eq. (1) results in:

$$H_X(s) = \frac{Y(s)}{X(s)} = \frac{\frac{1}{s}}{1 + \frac{1}{s}} = \frac{1}{1 + s} \quad (2)$$

It can be seen that for small frequencies ($s \rightarrow 0$) the input is replicated at the output, and for high frequencies ($s \rightarrow \infty$) a lowpass characteristic is shown. This $H_X(s)$ is also called the *signal transfer function* (STF).

The transfer function $H_E(s)$ for the quantization error $E(s)$ to the output is (now setting $X(s) = 0$)

$$Y(s) = E(s) - \frac{1}{s}Y(s) \rightarrow Y(s) \left(1 + \frac{1}{s}\right) = E(s). \quad (3)$$

It follows that

$$H_E(s) = \frac{Y(s)}{E(s)} = \frac{1}{1 + \frac{1}{s}} = \frac{s}{1 + s}. \quad (4)$$

It can be seen that the transfer function for the quantization error to the output shows a high pass response, for small frequencies the quantization error is thus suppressed, and the quantization noise is moved towards high frequencies! This $H_E(s)$ is also called the *noise transfer function* (NTF).

Let us now build a sampled modulator, as shown in fig. 5 (compare fig. 5 with fig. 3 — the ADC and DAC is removed as the whole signal flow is now in the digital domain). **The interesting part is that the bit width at the output y_i can be much less than the bit width at the input x_i !** In our implementation, we take 16 bit at the input, and only a single bit at the output.

The integrator block required in this sampled implementation can be realized by a simple accumulator, as shown in fig. 6. Checking the transfer function in the z -domain, we arrive at

$$Y(z) = X(z) + z^{-1}Y(z) \rightarrow Y(z)(1 - z^{-1}) = X(z). \quad (5)$$

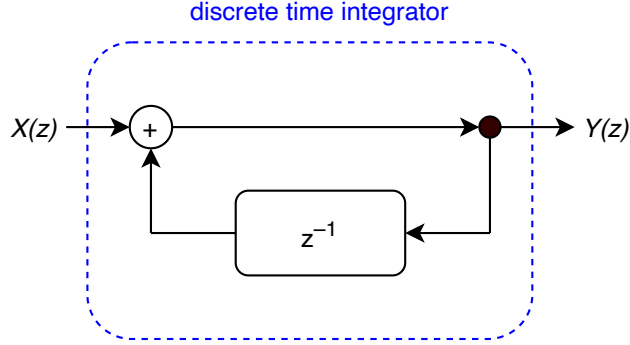


Figure 6: Discrete time integrator, consisting of an adder and a register.

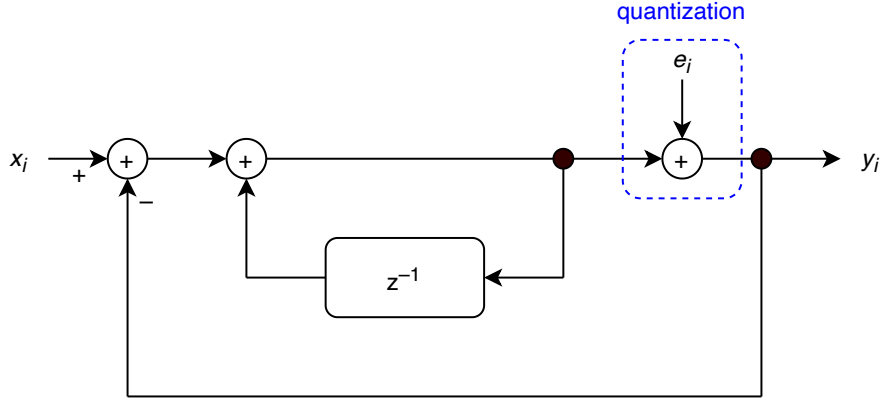


Figure 7: First-order digital delta-sigma modulator.

Thus it follows that

$$H_{\text{int}}(z) = \frac{Y(z)}{X(z)} = \frac{1}{1 - z^{-1}} \quad (6)$$

Let us now check whether the result of eq. (6) makes sense. We can transform from the z - into the s -domain by setting $z = e^{sT}$, with $T = 1/f_s$ being the sampling period. This results in

$$H_{\text{int}}(s) = \frac{1}{1 - e^{-sT}}.$$

Consequently, $H_{\text{int}}(s \rightarrow 0) = \infty$, so that corresponds correctly to the behavior of an integrator. The gain of the integrator drops at half the sampling frequency $f = 1/(2T)$ to $H_{\text{int}}(s = j2\pi(2T)^{-1}) = 1/2$.

So the complete discrete-time modulator can be redrawn as shown in fig. 7. The STF(z) can then be calculated to (we set $E(z) = 0$)

$$Y(z) = Y(z)z^{-1} - Y(z) + X(z) \rightarrow Y(z) [2 - z^{-1}] = X(z). \quad (7)$$

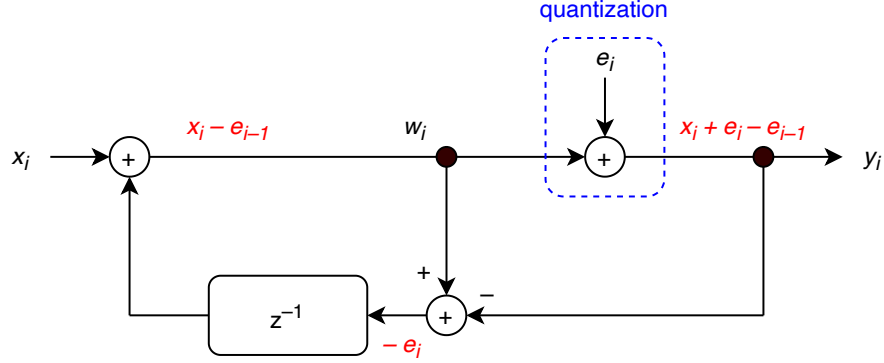


Figure 8: Error-feedback architecture.

$$\text{STF}(z) = \frac{1}{2 - z^{-1}} \quad (8)$$

For low frequencies ($z = 1$) this results in $\text{STF}(z = 1) = 1$. For high frequencies at $f_s/2$ ($z = -1$) this results in $\text{STF}(z = -1) = 1/3$. This makes sense, as for low frequencies the input signal is passed to the output, and for higher frequencies, there is some attenuation.

The NTF(z) we calculate to ($X(z) = 0$):

$$Y(z) = E(z) - \frac{Y(z)}{1 - z^{-1}} \rightarrow Y(z) \left(1 + \frac{1}{1 - z^{-1}} \right) = E(z) \quad (9)$$

$$\text{NTF}(z) = \frac{1 - z^{-1}}{2 - z^{-1}} \quad (10)$$

Again, for low frequencies this results in $\text{NTF}(z = 1) = 0$, and for high frequencies $\text{NTF}(z = -1) = 2/3$, showing the wanted high-pass shaping of the quantization error.

We could now directly implement the digital architecture of fig. 7, however, there exists an equivalent form that leads to even better implementation, called the *error feedback architecture*. This architecture is shown in fig. 8, and while implementation in an ADC is not favorable (an analog and a digital quantity would need to be subtracted leading to gain mismatch errors³), in the digital domain this leads to a very efficient design, as we will see soon.

The $\text{STF}(z)$ can be easily extracted from fig. 8 when setting $e_i = 0$ (and consequently $e_{i-1} = 0$):

$$\text{STF}(z) = 1 \quad (11)$$

In this architecture, the input signal is passed to the output without filtering.

The $\text{NTF}(z)$ is equally found directly by setting $x_i = 0$ and is given as

$$Y(z) = E(z) - E(z)z^{-1} = E(z)(1 - z^{-1}) \rightarrow \text{NTF}(z) = 1 - z^{-1} \quad (12)$$

³Remember, we have an ADC at the location where the quantization error is injected into the model.

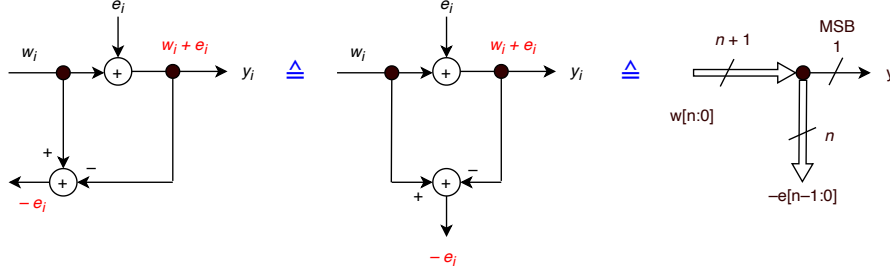


Figure 9: Arriving at the final digital implementation of the error-feedback architecture: just splitting a digital bus. This only works for **unsigned integers**, however.

Table 3: Example for 2 bit input bit-width

w (dec)	w (bin)	y (bin)	y (dec)	$-e$ (dec)	$-e$ (bin)
0	000	0	0	0	00
1	001	0	0	1	01
2	010	0	0	2	10
3	011	0	0	3	11
4	100	1	4	0	00
5	101	1	4	1	01
6	110	1	4	2	10
7	111	1	4	3	11

We can double-check this NTF by equating it for low frequencies, resulting in $\text{NTF}(z = 1) = 0$, and for high frequencies (with the maximum frequency at $f_s/2$), resulting in $\text{NTF}(z = -1) = 2$. As expected, we see the high pass shaping of the quantization error.

For a digital modulator, where the bit width is reduced, this architecture is easily realized, which can be seen by identifying the identities shown in fig. 9.

We can see that the error-feedback architecture can be realized by splitting a bus and just adding a register (for the delay) and an adder. It is important to note that this simple implementation works only if **unsigned integer** representation is used for the data! An important observation is that we can define the quantization error e_i as a negative number, so the feedback error $-e_i$ is a positive number!

In order to comprehend the functionality shown in fig. 9 let us walk through a simple example using $n = 2$. The resulting quantities of w , y , and e are shown in table 3. This example nicely shows that by subtracting the MSB from w for the output we get $-e$ from $w[\text{MSB} - 1 : 0]$, which is then fed back.

The full architecture of the digital implementation is shown in fig. 10. The input word X has a width of n bit, and the output Y is a 1 bit quantity which is easily converted to an analog voltage, e. g. by using a strong inverter, and the register also has a width of n bit.

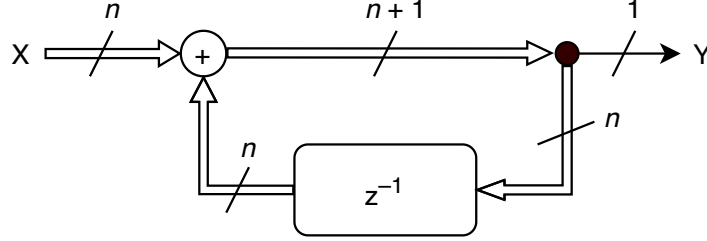


Figure 10: Delta-sigma error-feedback modulator of first order, with n -bit input (unsigned integer) and 1 bit output.

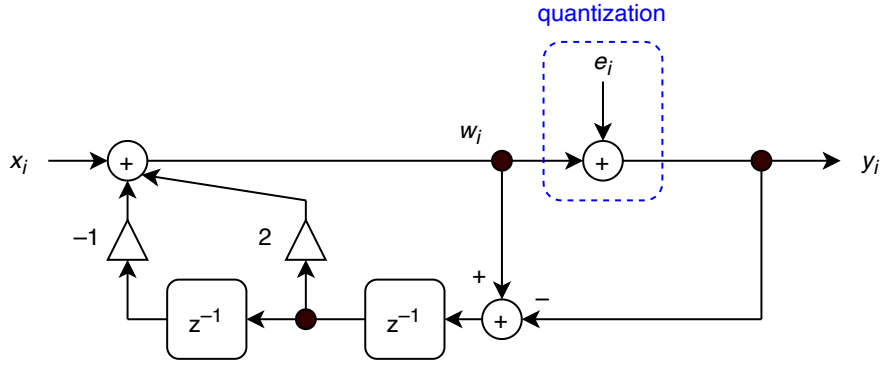


Figure 11: Signal flow-graph of a second-order delta-sigma modulator.

A second-order delta-sigma modulator can be constructed by changing the feedback filter from z^{-1} to $z^{-1}(2 - z^{-1})$, which is illustrated in fig. 11. The NTF(z) is given by

$$\text{NTF}(z) = 1 - z^{-1}(2 - z^{-1}). \quad (13)$$

The STF(z) is the same as for the first-order error-feedback type:

$$\text{STF}(z) = 1 \quad (14)$$

Generally, a higher-order modulator suppresses the quantization noise better at lower frequencies at the expense of larger noise at higher frequencies. A practical realization is depicted in fig. 12. Note that the 2 bit output of the first stage is transformed into a 1 bit output by a following first-order modulator.

Note that the first delta-sigma modulator can run at a lower sampling frequency than the second modulator, which saves power!

7 Digital design

We will use Verilog for the implementation of the digital circuits, specifically the Verilog-2005 flavor of the language (as this is supported by the synthesis tool Yosys). Here are pointers where you can find introductory material:

8 Exemplary implementation and verification scripts

8.1 Matlab/Octave code for architecture development

File arch/dac_design.m:

```
1 disp('-----');
2 disp('DAC Analyzer (c) 2021 Harald Pretl, IIC, JKU');
3 disp('-----');
4 disp('');
5 pkg load signal; % for 'rms', 'interp' and 'decimate'
6 clear;
7
8 % name of audio tracks
9 % -----
10 audio_file_in = 'testaudio.wav';
11 audio_file_out = 'testaudio_out.wav';
12
13 % parameters
14 % -----
15 n = 16;
16 % no of bits; we represent data as unsigned integer
17 osr = 128;
18 % oversampling ratio of delta-sigma
19 interp_method = 1;
20 % 0 = zero insertion
21 % 1 = zero-order hold
22 % 2 = first-order hold [not implemented]
23 % 3 = interpolation with sinc
24 sd_type = 2;
25 % 1 = first-order modulator
26 % 2 = second-order modulator
27 use_sine = false;
28 % select if sine or audio used for testing
29
30 % read test audio file, is signed .wav (+1...-1)
31 % -----
32 if use_sine == false
33     disp(strcat('Step 1: Read audio file, filename=', audio_file_in));
34     [track_in, fs_audio] = audioread(audio_file_in);
35     track_in = track_in';
36 else
37     disp('Step 1: Generate 441Hz sine tone with 60% fullscale');
38     fs_audio = 44100;
39     t_sin = (0:fs_audio-1)/fs_audio;
40     track_in = 0.6 * sin(2*pi * 441 * t_sin);
41     end % if
42
43 disp(strcat('Info: fs=', num2str(fs_audio), 'Hz'))
44 % remove dc
45 track_in = track_in - mean(track_in);
46 % scale track (signed float to unsigned int by adding offset)
47 u = round((track_in+1)*2^(n-1));
48
49 % derived parameters
50 fullscale = 2^n;
51 fs_out = fs_audio * osr; % Hz
52
53 % upsample by osr, insertion of zeros
54 % -----
55 if (interp_method == 0)
56     % interpolation by inserting zeros
57     u_samples = length(u);
58     u_up = zeros(1, u_samples*osr);
59     u_up(osr*(1:u_samples)) = u;
60 elseif (interp_method == 1)
61     % interpolation by zero-order hold
62     u_up = repelem(u, osr);
63 elseif (interp_method == 2)
64     % interpolation by first-order hold
65     % FIXME
66 elseif (interp_method == 3)
67     % interpolation by sinc
68     u_up = interp(u, osr);
69 end % if
70
71 % delta-sigma core
72 % -----
73 no_samples = length(u_up);
74 progress_step = round(no_samples/10);
75 reg1 = 0; reg2 = 0; reg3 = 0; c1 = 0;
76 out_sd = zeros(1, no_samples);
77
78 disp('Step 2: Perform delta-sigma processing...');
79
80 if (sd_type == 1) % first order modulator
81     for i=1:no_samples
82         % print a progress bar
83         if (rem(i, progress_step) == 0)
84             disp(strcat(' ', num2str(round(i/no_samples*100)), '%'));
85         end % if
86
87         out = u_up(i) + reg1; % 17b UINT
88         reg1 = mod(out, fullscale); % 16b UINT; cut off MSB and store
89         out_sd(i) = fix(out / fullscale); % 1b; MSB is output bit
90     end % for
```

```

91 end % if
92
93 if (sd_type == 2) % second order modulator
94     for i=1:no_samples
95         % print a progress bar
96         if (rem(i,progress_step) == 0)
97             disp(strcat(' ',num2str(round(i/no_samples*100)),'%'));
98         end % if
99
100         if (mod(i,4) == 0) ; % first modulator runs on fs/4
101             out1 = u_up(i) + 2*reg1 + (fullscale - reg2); % 18b UINT
102             reg2 = reg1; % 16b UINT; reg2 = reg1 * z^-1
103             reg1 = mod(out1,fullscale); % 16b UINT; cut off MSB/MSB-1 and store
104             c1 = fix(out1 / fullscale); % 2b UINT; MSB/MSB-1 is output bits
105         end % if
106         out2 = c1 + reg3; % 3b UINT
107         reg3 = mod(out2,4); % 2b UINT; cut off MSB and store
108         out_sd(i) = fix(out2 / 4); % 1b; MSB is output bit
109     end % for
110 end % if
111
112 disp('...done');
113
114 % decimate audio for playback, and scale according to input
115 % -----
116 % we use a halfband filter approach to decimate
117 filt1 = fir1(osr, 1/osr);
118 filt2 = fir1(osr/4, 4/osr);
119 temp1 = fftfilt(filt1, out_sd);
120 temp2 = temp1(1:2:length(temp1));
121 temp3 = fftfilt(filt1, temp2);
122 temp4 = temp3(1:2:length(temp3));
123 temp5 = fftfilt(filt2, temp4);
124 out_sd_dec = temp5(1:(osr/4):length(temp5));
125 % now get back to audio track format
126 track_out = (out_sd_dec - mean(out_sd_dec)); % remove dc
127 track_out = track_out * rms(track_in)/rms(track_out); % balance loudness
128 track_out = min(0.999,track_out); % clip max
129 track_out = max(-0.999,track_out); % clip min
130 track_out = track_out * rms(track_in)/rms(track_out); % balance after clip
131
132 % write resulting audio track
133 % -----
134 disp(strcat('Step 3: Write audio file, filename=',audio_file_out));
135 audiowrite(audio_file_out,track_out,'fs_audio');
136
137 % calculate SNR
138 % -----
139 track_out(1:length(track_out)-1) = track_out(2:length(track_out));
140 sqnr_sd = 20*log10(rms(track_out)/rms(track_in-track_out));
141 disp(strcat('Info: SQNR=',num2str(sqnr_sd),'dB'));
142
143 % byebye
144 disp('');
145 disp('Done, bye!');

```

8.2 Verilog code for delta-sigma modulator

File dig/rtl/audiodac.v:

```

1  /*
2  * AUDIODAC -- 16b Delta-Sigma Modulator with Single-Bit Output
3  *
4  * (c) 2021-2022 Harald Pretl (harald.pretl@jku.at)
5  * Johannes Kepler University Linz, Institute for Integrated Circuits
6  *
7  * This delta-sigma modulator includes an input-side FIFO for easy connection
8  * to a host system, volume control in 6dB steps, and configurable OSR
9  * (32/64/128/256). The modulator can be configured to first or second order.
10 *
11 * The clock for the modulator is required to be clk = audio_sample_rate * OSR
12 * e.g. for 44.1kHz and OSR=128 then clk=5.6448MHz
13 *
14 * The audio input data (fifo_i) must be 16b signed INT!
15 *
16 * In order to write into the FIFO:
17 * (1) assert the data at fifo_i[15:0] (with fifo_rdy_i=0)
18 * (2) signal that data is ready with a transition of fifo_rdy_i=0->1
19 * (3) wait that data is accepted by waiting for the transition fifo_ack_o=0->1
20 * (4) de-assert fifo_rdy_i=1->0 (this is followed by fifo_ack_o=1->0)
21 *
22 * Data is only accepted if the FIFO is not full (fifo_full_o=0). Using fifo_full_o
23 * and fifo_empty_o an interrupt service routine (ISR) can be constructed:
24 * (1) fifo_empty_o=0->1 triggers the ISR
25 * (2) the ISR writes data into the FIFO until it is full (fifo_full_o=0->1)
26 * (3) exit the ISR
27 *
28 * Ideas for future extension:
29 * (1) Implement a higher-order (e.g. 5th order) modulator.
30 * (2) Potentially substitute register-based FIFO by a dual-port RAM.
31 */
32
33 `default_nettype none
34 `ifndef __AUDIODAC__

```

```

35 `define __AUDIODAC__
36
37 `ifdef SIM_PYTHON
38     `include "../dig/rtl/iic_dsmod.v"
39     `include "../dig/rtl/iic_fifo.v"
40     `include "../dig/rtl/iic_sinegen.v"
41 `else
42     `include "iic_dsmod.v"
43     `include "iic_fifo.v"
44     `include "iic_sinegen.v"
45 `endif
46
47 module audiodac (
48     // FIFO interface
49     input      [15:0]  fifo_i,      // data is signed INT
50     input      fifo_rdy_i,
51     output wire      fifo_ack_o,
52     output wire      fifo_full_o,
53     output wire      fifo_empty_o,
54     // housekeeping
55     input      rst_n_i,
56     input      clk_i,
57     // configuration bits
58     input      mode_i,      // 0 = 1st, 1 = 2nd order SD-mod
59     input      [3:0]  volume_i, // 0 = off, 1 = min volume, 15 = max volume
60     input      [1:0]  osr_i,  // 0 = 32; 1 = 64, 2 = 128, 3 = 256
61     // DS-modulator outputs
62     output wire      ds_o,      // single-bit SD-modulator output
63     output wire      ds_n_o,    // plus the complementary output
64     // test modes
65     input      tst_fifo_loop_i,
66     input      tst_sinegen_en_i,
67     input      [4:0]  tst_sinegen_step_i
68 );
69
70 parameter SINE_GENERATOR_AMPL = 0.5;
71 parameter SINE_GENERATOR_LUT_SIZE = 6;
72
73 wire      [15:0]  dsmod_data_i, audio, audio_fifo, audio_sinegen;
74 wire      audio_rd;
75 assign     audio = tst_sinegen_en_i ? audio_sinegen : audio_fifo;
76 // change 16-bit signed audio data to unsigned data
77 assign     dsmod_data_i = audio + {1'b1,{15{1'b0}}};
78
79 iic_fifo
80     fifo0 (
81         .fifo_indata_i(fifo_i),
82         .fifo_indata_rdy_i(fifo_rdy_i),
83         .fifo_indata_ack_o(fifo_ack_o),
84         .fifo_full_o(fifo_full_o),
85         .fifo_empty_o(fifo_empty_o),
86         .fifo_outdata_o(audio_fifo),
87         .fifo_outdata_rd_i(audio_rd),
88         .rst_n_i(rst_n_i),
89         .clk_i(clk_i),
90         .tst_fifo_loop_i(tst_fifo_loop_i)
91     );
92
93 iic_dsmod
94     dsmod0 (
95         .data_i(dsmod_data_i),
96         .data_rd_o(audio_rd),
97         .ds_o(ds_o),
98         .ds_n_o(ds_n_o),
99         .rst_n_i(rst_n_i),
100        .clk_i(clk_i),
101        .mode_i(mode_i),
102        .scale_i(volume_i),
103        .osr_i(osr_i)
104    );
105
106 iic_sinegen
107     #(16, SINE_GENERATOR_LUT_SIZE, SINE_GENERATOR_AMPL)
108     sinegen0 (
109         .data_o(audio_sinegen),
110         .data_rd_i(audio_rd),
111         .rst_n_i(rst_n_i),
112         .clk_i(clk_i),
113         .tst_sinegen_en_i(tst_sinegen_en_i),
114         .tst_sinegen_step_i(tst_sinegen_step_i)
115     );
116
117 endmodule // audiodac
118
119 `endif
120 `default_nettype wire

```

File dig/rtl/audiodac_dsmod_template.v:

```

1 /*
2 * IIC_DSMOD -- Delta-Sigma Modulator (1st/2nd Order) with Single-Bit Output
3 *
4 * (c) 2021-2022 Harald Pretl (harald.pretl@jku.at)
5 * Johannes Kepler University Linz, Institute for Integrated Circuits
6 *
7 * Ports:
8 *     data_i      ... input data (must be UINT, default 16b)

```

```

9  *   data_rd_o    ... fetch next input data word (data feed-in driven
10 *               by DS-modulator)
11 *   ds_o, ds_n_o ... output bitstream (complimentary outputs)
12 *
13 *   rst_n_i      ... reset (low active)
14 *   clk_i        ... clock of ds-modulator (must be OSR * input_data_rate),
15 *               output bitstream rate = OSR * input_data_rate
16 *
17 *   mode_i       ... select order of modulator (0 = 1st, 1 = 2nd)
18 *   scale_i      ... scaling (attenuation) of input data in -6dB steps
19 *               (0 = 0dB, 1 = -6dB, 2 = -12dB, ..., 15 = off)
20 *   osr_i        ... oversampling ratio (OSR), 0=32/1=64/2=128/3=256 is supported
21 */
22
23 `default_nettype none
24 `ifndef __IIC_DSMOD__
25 `define __IIC_DSMOD__
26
27 module iic_dsmod (
28     input          [15:0]      data_i,
29     output wire    data_rd_o,
30     output reg     ds_o,
31     output wire    ds_n_o,
32
33     input          rst_n_i,
34     input          clk_i,
35
36     input          mode_i,
37     input          [3:0]    scale_i,
38     input          [1:0]    osr_i
39 );
40
41     // Your code goes here!
42
43 endmodule // iic_dsmod
44
45 `endif
46 `default_nettype wire

```

File dig/rtl/audiodac_fifo.v:

```

1  /*
2  *   IIC_FIFO -- Configurable FIFO based on ring-buffer
3  *
4  *   (c) 2021-2022 Harald Pretl (harald.pretl@jku.at)
5  *   Johannes Kepler University Linz, Institute for Integrated Circuits
6  *
7  *   Ports:
8  *   fifo_indata_i    ... input data
9  *   fifo_indata_rdy_i ... indicates that data is ready from source
10 *   fifo_indata_ack_o ... signals to source that datum has been taken into FIFO
11 *
12 *   fifo_full_o      ... indicates a full FIFO, no further datum will be
13 *                       acknowledged into FIFO
14 *   fifo_empty_o      ... indicates an empty FIFO, if datum is read from an empty
15 *                       FIFO then the last datum will be output
16 *
17 *   fifo_outdata_o    ... output data
18 *   fifo_outdata_rd_o ... the next output datum is selected (if FIFO is not empty)
19 *
20 *   rst_n_i          ... reset (low active)
21 *   clk_i            ... clock of FIFO
22 *
23 *   tst_fifo_loop_i  ... enables a test mode where the data in the FIFO is looped
24 *                       at the output
25 */
26
27 `default_nettype none
28 `ifndef __IIC_FIFO__
29 `define __IIC_FIFO__
30
31 module iic_fifo (
32     input          [15:0]    fifo_indata_i,
33     input          fifo_indata_rdy_i,
34     // note that fifo_indata_i and fifo_indata_rdy_i could originate from
35     // an asynchronous clk domain, so potentially needs to be synchronized
36     // use FIFO_ASYNC=1 to select the proper behaviour
37     output reg     fifo_indata_ack_o,
38     output wire    fifo_full_o,
39     output wire    fifo_empty_o,
40     output wire [15:0] fifo_outdata_o,
41     input          fifo_outdata_rd_i,
42     input          rst_n_i,
43     input          clk_i,
44     input          tst_fifo_loop_i
45 );
46
47     // Your code goes here!
48
49 endmodule // iic_fifo
50
51 `endif
52 `default_nettype wire

```

File dig/rtl/audiodac_sinegen.v:

```

1  /*
2  * IIC_SINEGEN -- Simple Sine Generator based on a hardcoded 16b LUT
3  *
4  * (c) 2021-2022 Harald Pretl and Jakob Ratschenberger (harald.pretl@jku.at)
5  * Johannes Kepler University Linz, Institute for Integrated Circuits
6  *
7  * Ports:
8  *   data_o          ... output data of sine generator (BW)
9  *   data_rd_i       ... advances the generator to the next output value,
10 *                     controlled by the step size
11 *
12 *   rst_n_i         ... reset (low active)
13 *   clk_i           ... clock of sine generator (pos. edge)
14 *
15 *   tst_sinegen_en_i ... enables the sine generator; when disabled then 0 is
16 *                     output, and the generator is turned off
17 *   tst_sinegen_step_i ... controls the step size per read (1 = use every ROM
18 *                     value, 2 = use every 2nd ROM entry, etc); the higher the
19 *                     step value, the faster the output sine
20 */
21
22 `default_nettype none
23 `ifndef __IIC_SINEGEN__
24 `define __IIC_SINEGEN__
25
26 module iic_sinegen (
27     output wire signed [15:0] data_o,
28     input data_rd_i,
29     input rst_n_i,
30     input clk_i,
31
32     input tst_sinegen_en_i,
33     input [3:0] tst_sinegen_step_i
34 );
35
36     // Your code goes here!
37
38 endmodule // iic_sinegen
39
40 `endif
41 `default_nettype wire

```

8.3 Python3-driven Verilog test-bench

We are using a Python3-based test-bench for the verification of the Verilog code, where a sinusoidal signal, filtered white noise, or an actual audio signal can be used as a stimulus for the digital simulation. The Python3 script sets the parameters for the Verilog testbench, and evaluates the simulation output. We are providing the top-level verification testbench to check the final implementation, however, it is good practice to implement unit tests for each module to verify the component implementations before trying to validate the overall design.

To run a top-level simulation in this test-bench change into the **py** directory and run:

```
> python3 audiodac_tb.py
```

File `py/audiodac_tb.py`:

```

1  """
2  AUDIODAC_TB
3
4  (c) 2022 Jakob Ratschenberger
5  Johannes Kepler University Linz, Institute for Integrated Circuits
6
7  Python script to controll and perform the simulation of the AUDIODAC.
8
9  Implemented functionality:
10 - Setting the AUDIODAC configurations (OSR, MODE, VOLUME)
11 - Setting the AUDIODAC input data
12   -- Four sets are available:
13     + Filtered White Gaussian Noise
14     + Sine with 10kHz
15     + All zeros
16     + Sound of an bell
17 - Data visualisation
18   -- Plotting the input data in time and frequency domain
19   -- Plotting the output data in frequency domain
20 - Generating .wav of the AUDIODAC output for the sound input
21   -- Resamples the AUDIODAC output stream to 44100Hz
22
23 """

```



```

24
25
26 import os
27 import numpy as np
28 import scipy.signal as signal
29 from scipy.io import wavfile
30 import matplotlib.pyplot as plt
31 import AudioDACSimsVals as simvals
32 import AudioDACSimsResults as simresults
33
34 fs = 44100 #Input sampling rate
35
36 BW = 16 #Bitwidth
37
38 #Request the simulation parameters
39
40 #Set the Oversampling Rate (OSR)
41 OSR = int(input("SIM_OS: (32/64/128/256) "))
42 if OSR not in [32,64,128,256]:
43     raise ValueError("OSR {} not supported!".format(OSR))
44
45 #Convert the entered OSR in the desired range of the AUDIODAC
46 SIM_OS = 0
47 if OSR == 64:
48     SIM_OS = 1
49 elif OSR == 128:
50     SIM_OS = 2
51 elif OSR == 256:
52     SIM_OS = 3
53
54 #Set the mode of the AUDIODAC
55 #Mode 0 => 1st order DSMOD
56 #Mode 1 => 2nd order DSMOD
57 SIM_MODE = int(input("SIM_MODE: (0/1) "))
58 if SIM_MODE not in [0,1]:
59     raise ValueError("SIM_MODE {} not supported!".format(SIM_MODE))
60
61 #Set the volume
62 SIM_VOLUME = int(input("SIM_VOLUME: (0 = 0dB, 1 = -6dB, 15 = off) "))
63 if SIM_VOLUME not in [_ for _ in range(16)]:
64     raise ValueError("SIM_VOLUME {} not supported!".format(SIM_VOLUME))
65
66
67 #Set the desired test set
68 TestSet = int(input("Which testset?
69 (1) Filtered WGN
70 (2) Sine (10kHz)
71 (3) Zeros
72 (4) Sound
73 Your selection: "))
74
75 if TestSet not in [1,2,3, 4]:
76     raise ValueError("Testset not supported!")
77
78 #If the test set isn't the sound, ask for the number of simulated samples
79 if TestSet not in [4]:
80     nSamples = int(input("Number of samples ([44,441000]): "))
81     #Set a lower and upper bound
82     #Lower bound to prevent to less output samples
83     #Upper bound to prevent to long simulation
84     if nSamples < 44 or nSamples > 10*44100:
85         raise ValueError("Number of samples not supported!")
86
87 #Set the input data
88 if TestSet == 1:
89     test_data = np.random.randint(-2**(BW-1), 2**(BW-1)-1,nSamples)
90     lowpass = signal.firwin(100, fs/4, fs=fs)
91     test_data = signal.lfilter(lowpass, 1, test_data)
92 elif TestSet == 2:
93     test_data = 0.5*2**(BW-1)*np.sin(2*np.pi*10000/fs * np.arange(nSamples))
94 elif TestSet == 3:
95     test_data = 0*np.arange(nSamples)
96 elif TestSet == 4:
97     sound_sample_rate, test_data = wavfile.read("Bell.wav")
98     nSamples = len(test_data)
99
100
101 #Calc. the spectrum of the test data
102 freq_test_data = np.fft.fft(test_data, norm="forward")
103 f = np.fft.fftfreq(freq_test_data.size, 1/fs)
104
105 #Plot the test data in time domain
106 fig, ax = plt.subplots()
107 fig.suptitle(r'Test data')
108 ax.plot(test_data, 'b')
109 ax.set_xlabel(r'n')
110 ax.set_ylabel(r'$x_{Test}[n]$')
111 ax.grid(True)
112 plt.draw()
113
114 #Plot the spectrum of the test data
115 fig, ax = plt.subplots()
116 fig.suptitle(r'Spectrum of the test data')
117 x = np.fft.fftshift(f)
118 y = np.abs(np.fft.fftshift(freq_test_data))
119 ax.plot(x,y, 'b')
120 ax.set_xlabel(r'$f$, /, \, Hz$')
121 ax.set_ylabel(r'$\mid \mathrm{fft} \mid \left(x_{Test}[n] \right) \mid \mathrm{mid}$ ')
122 ax.grid(True)

```

```

123 plt.draw()
124
125 #Setup the simulation values
126 mySimVals = simvals.AudioDACSimVals(SIM_MODE, SIM_OSR, SIM_VOLUME, test_data)
127
128 #Generate the necessary files for the sim
129 mySimVals.genSimFiles()
130
131 #Run the verilog simulation
132 os.system("iverilog -g 2005 -o AUDIODAC_PY_TB -c file_list.txt")
133 os.system("vvp AUDIODAC_PY_TB")
134
135 #Set the file name of the simulation result
136 mySimResults = simresults.AudioDACSimResults("verilog_bin_out.txt")
137
138 #Read the simulation result
139 data = mySimResults.getSIM_Result()
140
141 data = (data)*2-1 #Merge the poitive and negative AUDIODAC output
142
143 data = data*2**(BW-1) #Scale the data to input range
144
145 #Calc. the spectrum of the AUDIODAC output stream
146 freq_data = np.fft.fft(data, norm="forward")
147 f = np.fft.fftfreq(freq_data.size,d=1/(fs*OSR))
148
149 #Plot the spectrum of the output data
150 fig, ax = plt.subplots()
151 fig.suptitle(r'Spectrum of the output data')
152 x = np.fft.fftshift(f)
153 y = np.abs(np.fft.fftshift(freq_data))
154 ax.plot(x,y, 'b')
155 ax.set_xlabel(r'$f\backslash,\backslash,\text{Hz}$')
156 ax.set_ylabel(r'$\text{mid } \mathrm{fft}\backslash\left(x_{\text{out}}\backslash\right)\backslash\text{mid}$')
157 ax.grid(True)
158 plt.draw()
159
160 #Get the output spectrum in the range 0 to fs/2
161 indx_fs = round(len(freq_data)/(OSR*2))
162 freq_data2 = freq_data[0:indx_fs-1]
163 f = f[0:indx_fs-1]
164
165 #Plot the output spectrum in the range 0 to fs/2
166 fig, ax = plt.subplots()
167 fig.suptitle(r'Zoomed spectrum of the output data')
168 x = f
169 y = np.abs(freq_data2)
170 ax.plot(x,y,'b')
171 ax.set_xlabel(r'$f\backslash,\backslash,\text{Hz}$')
172 ax.set_ylabel(r'$\text{mid } \mathrm{fft}\backslash\left(x_{\text{out}}\backslash\right)\backslash\text{mid}$')
173 ax.grid(True)
174 plt.draw()
175
176
177 #If the test set was the sound file
178 if TestSet == 4:
179
180     #Resample the AUDIODAC output stream to fs by using Fourier method
181     nSamples = round(len(data)/OSR)
182     data = signal.resample(data,int(nSamples))
183
184     #Write the output to a .wav file
185     wavfile.write("AudioDAC.wav",fs, data.astype(np.int16))
186
187     #Plot the waveform
188     fig, ax = plt.subplots()
189     fig.suptitle(r'Waveform of the output data')
190     ax.plot(data)
191     ax.grid(True)
192     plt.draw()
193
194 #Show all figures at the end
195 plt.show()

```

File dig/rtl/audiodac_tb.m:

```

1 /*
2  * AUDIODAC TESTBENCH -- 16b Delta-Sigma Modulator with Single-Bit Output
3  *
4  * (c) 2021-2022 Harald Pretl (harald.pretl@jku.at)
5  * Johannes Kepler University Linz, Institute for Integrated Circuits
6  *
7  * This is the testbench for audiodac.v
8  */
9
10 `timescale 10ns / 1ns
11
12 `include "../rtl/audiodac.v"
13
14 `ifndef SIM_LONG
15     `ifndef SIM_SHORT
16         `define SIM_SHORT
17     `endif
18 `endif
19
20 module audiodac_tb;

```

```

21
22 localparam SIM_MODE = 2;
23 localparam SIM_OSR = 2;
24 localparam SIM_VOLUME = 15;
25
26 // housekeeping for getting data in
27 `ifdef SIM_LONG
28     localparam DATA_SAMPLES = 44100*10;
29 `endif
30 `ifdef SIM_SHORT
31     localparam DATA_SAMPLES = 44100*1;
32 `endif
33
34 reg [15:0] DATA_IN[0:DATA_SAMPLES-1]; // large memory to hold the audio
35 reg DATA_OUT[0:(DATA_SAMPLES*(32*2^SIM_OSR)-1)]; // output results written to file
36 integer DATA_IN_CTR = 0;
37 integer DATA_OUT_CTR = 0;
38
39 // configuration bits
40 reg MODE = SIM_MODE;
41 reg [1:0] OSR = SIM_OSR;
42 reg [3:0] VOLUME = SIM_VOLUME;
43
44 // inputs
45 reg RESET_N = 1'b0;
46 reg CLK = 1'b0;
47 reg TST_FIFO_LOOP = 1'b0;
48 reg TST_SINEGEN_EN = 1'b1;
49 reg [4:0] TST_SINEGEN_STEP = 5'd2;
50
51 // outputs
52 wire FIFO_FULL, FIFO_EMPTY, FIFO_ACK, DS_OUT, DS_OUT_N;
53
54 // instantiate DUT
55 audiodac dac (
56     .fifo_i(DATA),
57     .fifo_rdy_i(DATA_RDY),
58     .fifo_ack_o(FIFO_ACK),
59     .fifo_full_o(FIFO_FULL),
60     .fifo_empty_o(FIFO_EMPTY),
61     .rst_n_i(RESET_N),
62     .clk_i(CLK),
63     .mode_i(MODE),
64     .volume_i(VOLUME),
65     .osr_i(OSR),
66     .ds_o(DS_OUT),
67     .ds_n_o(DS_OUT_N),
68     .tst_fifo_loop_i(TST_FIFO_LOOP),
69     .tst_sinegen_en_i(TST_SINEGEN_EN),
70     .tst_sinegen_step_i(TST_SINEGEN_STEP)
71 );
72
73
74 // make a clock
75 always #1 CLK = ~CLK;
76
77 // handle FIFO input data
78 reg [15:0] DATA = 16'b0;
79 reg DATA_RDY = 1'b0;
80 reg WAIT_FOR_EMPTY = 1'b0; // flag to signal a wait for the next write burst
81
82 always @(negedge CLK) begin
83
84     // provide input data
85     if (!DATA_RDY && !FIFO_FULL && !FIFO_ACK && !WAIT_FOR_EMPTY && RESET_N) begin
86         // option 1: just use a simple counter as data
87         // #1 DATA <= DATA + 1;
88
89         // option 2: use data from file
90         DATA <= DATA_IN[DATA_IN_CTR];
91         DATA_IN_CTR = DATA_IN_CTR + 1;
92
93         // signal to FIFO that data is ready
94         DATA_RDY <= 1'b1;
95
96         // no more input data left? write result and exit
97         if (DATA_IN_CTR == DATA_SAMPLES) begin
98             $writememh("verilog_bin_out.txt", DATA_OUT);
99             $finish;
100         end
101     end
102
103     // de-assert data_rdy when data transfer ack'd by FIFO
104     if (FIFO_ACK) DATA_RDY <= 1'b0;
105
106     // The FIFO data write-in is done in a bursty nature, like a
107     // host system would likely do. When the FIFO is empty we write
108     // until the FIFO is full, then we wait for the FIFO to get
109     // empty again--then we do another bursty data transfer. This
110     // is meant to put less burden on the host system, so just an
111     // interrupt service routine is required, no constant polling
112     // of the data_rdy line, instead fifo_empty can trigger the ISR.
113
114     // stall data write-in when FIFO is already full
115     if (FIFO_FULL) WAIT_FOR_EMPTY <= 1'b1;
116
117     // if FIFO is empty re-engage with data write-in
118     if (FIFO_EMPTY) WAIT_FOR_EMPTY <= 1'b0;
119

```

```

120 // store simulation result into memory for later dump
121 if (RESET_N) begin
122     DATA_OUT[DATA_OUT_CTR] <= DS_OUT;
123     DATA_OUT_CTR = DATA_OUT_CTR + 1;
124 end
125 end
126
127 // here is all the initiliazation work for the simulation
128 initial begin
129     // print out simulation state
130     $display("-----");
131     `ifdef SIM_SHORT
132         $display("Running short (1s) simulation.");
133     `elsif SIM_LONG
134         $display("Running long (10s) simulation.");
135     `endif
136     $display("-----");
137     $display("SIM MODE   = ", SIM_MODE);
138     $display("SIM_OSR    = ", SIM_OSR);
139     $display("SIM_VOLUME = ", SIM_VOLUME);
140     $display("-----");
141
142     // read in the testdata into the memory
143     `ifdef SIM_LONG
144         $readmemh("../ver/verilog_testaudio_10s.txt", DATA_IN);
145     `elsif SIM_SHORT
146         $readmemh("../ver/verilog_testaudio_1s.txt", DATA_IN);
147     `endif
148
149     // dump signals into VCD for debug
150     `ifdef SIM_SHORT
151         $dumpfile("audiodac_tb.vcd");
152         $dumpvars;
153     `endif
154
155     // de-assert reset
156     #5 RESET_N = 1'b1;
157
158     // provide an online output for tracking sim progress
159     //$monitor("At time %t, ds_out = %h", $time, DS_OUT);
160 end
161
162 endmodule // audiodac_tb

```

9 Optional work packages

Depending on the progress of the work different enhancements can be made to the basic delta-sigma modulator.

9.1 Test modes

To test the design at various stages of a product life cycle, different test modes would be helpful. Ideas for different test modes are:

- Static setting of the digital output signals to 0 and 1.
- Generation of a test signal (e. g. a sine) with different frequencies and amplitudes to allow testing without feeding-in data.

9.2 Wishbone interface

Instead of a custom-made FIFO interface, a Wishbone-compatible interface can be designed. Documentation about the Wishbone IP bus can be found on the internet at https://cdn.opencores.org/downloads/wbspec_b4.pdf.

9.3 Higher-order modulator

The proposed first- and second-order modulator shows good results, however, a higher-order modulator would perform better.