KVIC 336.007
16 b Delta-Sigma Audio DAC

© Harald Pretl
Institute for Integrated Circuits, Johannes Kepler University

v0.3 WS2022/23

Contents

Table 1: Specification and target performance

Specification	Value	Unit
Audio data input bit width (signed INT)	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 structure[1] to simplify the design, and will include a FIFO towards a host system[2]. 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.

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

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

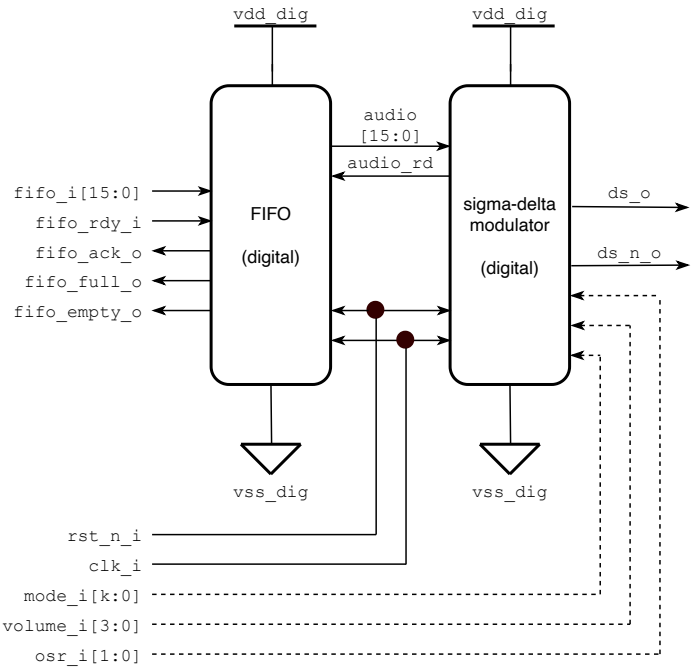

Figure 1: Audio-DAC high-level architecture.

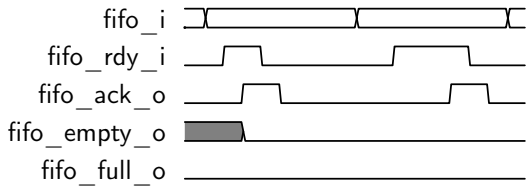

Figure 2: Host I/F timing.

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
`fifo_i[15:0]`	Digital audio data input into FIFO, signed INT	dig-I
`fifo_rdy_i`	Digital audio data ready	dig-I
`fifo_ack_o`	Digital audio data accepted into FIFO	dig-O
`fifo_full_o`	Indicator that FIFO is full, no data is loaded into FIFO	dig-O
`fifo_empty_o`	Indicator that FIFO is empty	dig-O
`rst_n_i`	Reset (active low)	dig-I
`clk_i`	Audio clock (OSR × audio input rate)	dig-I
`mode_i[k:0]`	Selection if first-order (`mode[0]=0`) or second-order modulation (`mode[0]=1`) *(implementation optional; mode[k:1] for other modes or test modes)*	dig-I
`volume_i[3:0]`	Volume control (`0` = no attenuation, `1` = −6 dB, `2` = −12 dB, ..., `15` = turn output off) *(implementation optional)*	dig-I
`osr_i[1:0]`	Oversampling ratio (OSR), `osr=0` is 32, `osr=1` is 64, `osr=2` is 128, `osr=3` is 256 *(implementation optional)*	dig-I
`vdd_dig`	Digital block supply voltage (1.8 V)	VDD
`vss_dig`	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.

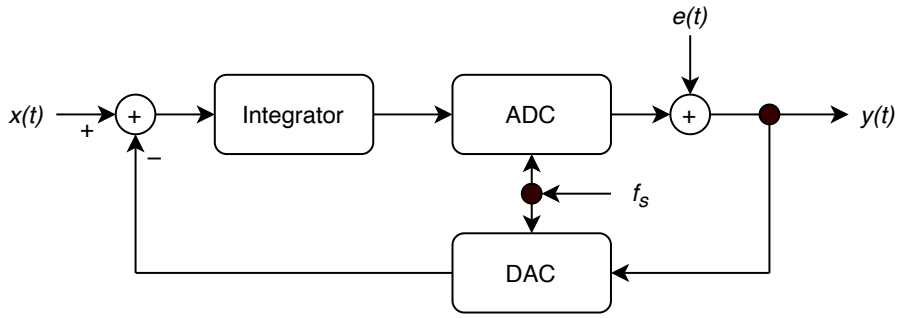

Figure 3: A simple analog-digital delta-sigma modulator architecture.

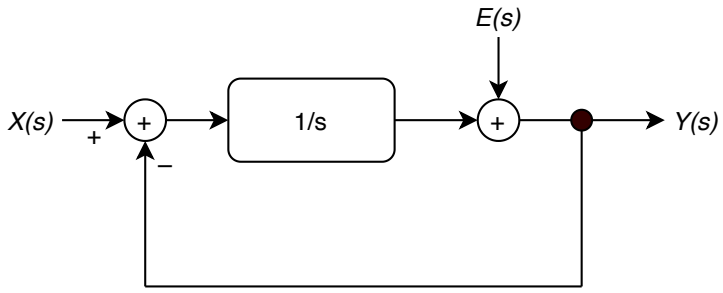

Figure 4: s-domain model of the simple delta-sigma modulator.

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 ($f_s \gg f_{\mathrm{in,max}}$). 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 $E(s) = 0$):

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

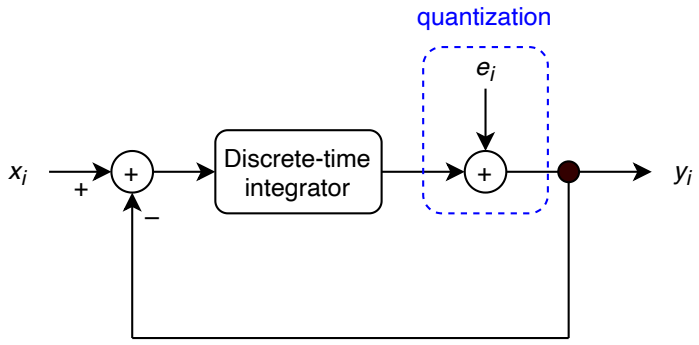

Figure 5: Discrete time version of the digital delta-sigma modulator.

Further manipulation of eq. (1) results in:

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

It can be seen that for small frequencies ($s \to 0$) the input is replicated at the output, and for high frequencies ($s \to \infty$) 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 $X(s) = 0$)

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

It follows that

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

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

$$Y(z) = X(z) + z^{-1}Y(z) \to Y(z)\left(1-z^{-1}\right) = X(z). \tag{5}$$

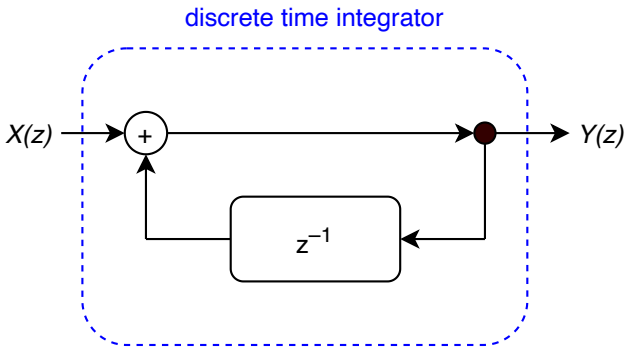

Figure 6: Discrete time integrator, consisting of an adder and a register.

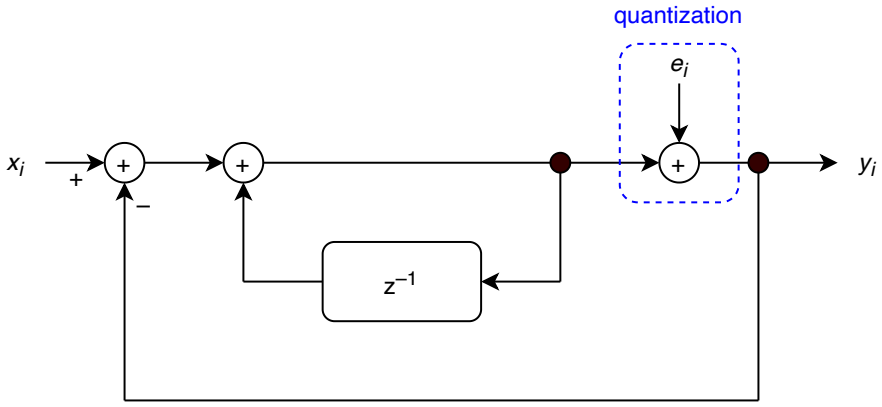

Figure 7: First-order digital delta-sigma modulator.

Thus it follows that

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

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, $H_{\text{int}}(s \to 0) = \infty$, 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 $H_{int}(s = j2\pi(2T)^{-1}) = 1/2$.

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

$$Y(z) = Y(z)z^{-1} - Y(z) + X(z) \to Y(z)\left[2 - z^{-1}\right] = X(z). \tag{7}$$

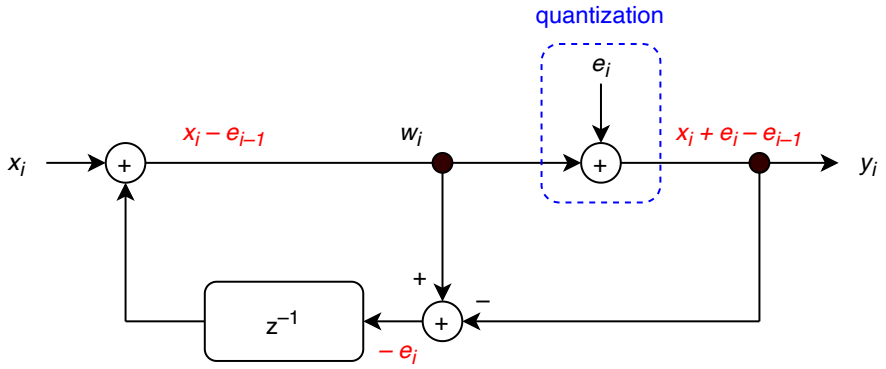

Figure 8: Error-feedback architecture.

$$\mathrm{STF}(z) = \frac{1}{2 - z^{-1}} \tag{8}$$

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

The $\mathrm{NTF}(z)$ we calculate to ($X(z) = 0$):

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

$$\mathrm{NTF}(z) = \frac{1 - z^{-1}}{2 - z^{-1}} \tag{10}$$

Again, for low frequencies this results in $\mathrm{NTF}(z = 1) = 0$, and for high frequencies $\mathrm{NTF}(z = -1) = 2/3$, 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 errors[3]), in the digital domain this leads to a very efficient design, as we will see soon.

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

$$\mathrm{STF}(z) = 1 \tag{11}$$

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

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

$$Y(z) = E(z) - E(z)z^{-1} = E(z)\left(1 - z^{-1}\right) \rightarrow \mathrm{NTF}(z) = 1 - z^{-1} \tag{12}$$

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

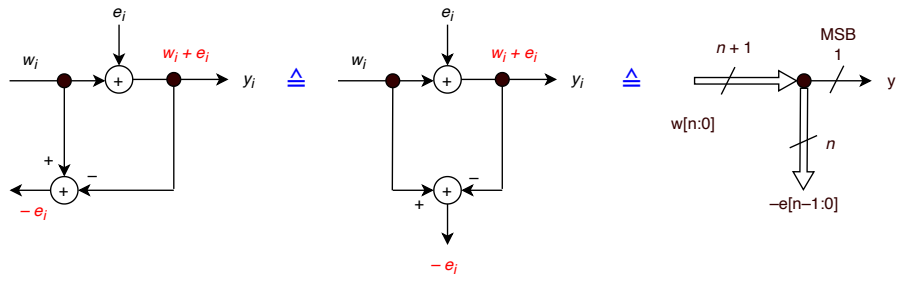

Figure 9: Arriving at the final digital implementation of the error-feedback architecture: just splitting a digital bus. This only works for **unsigned integers**, however.

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 NTF($z = 1$) $= 0$, and for high frequencies (with the maximum frequency at $f_s/2$), resulting in 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 $w[MSB - 1 : 0]$, 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.

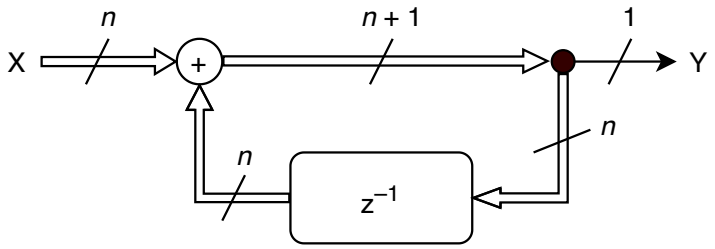

Figure 10: Delta-sigma error-feedback modulator of first order, with n-bit input (unsigned integer) and 1 bit output.

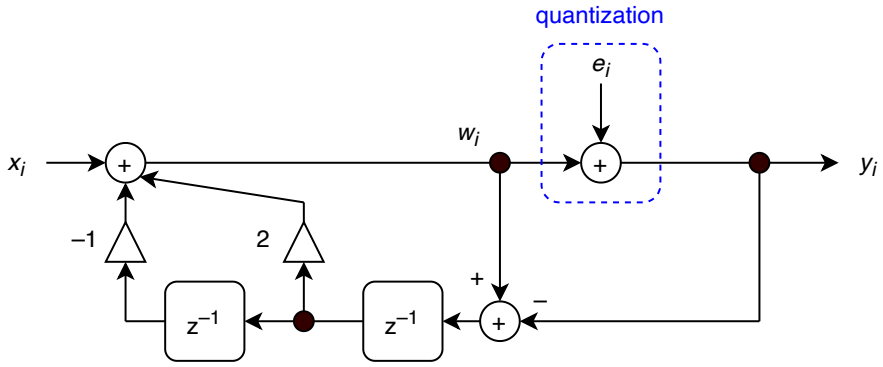

Figure 11: Signal flow-graph of a second-order delta-sigma modulator.

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

$$\mathrm{NTF}(z) = 1 - z^{-1}\left(2 - z^{-1}\right). \tag{13}$$

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

$$\mathrm{STF}(z) = 1 \tag{14}$$

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:

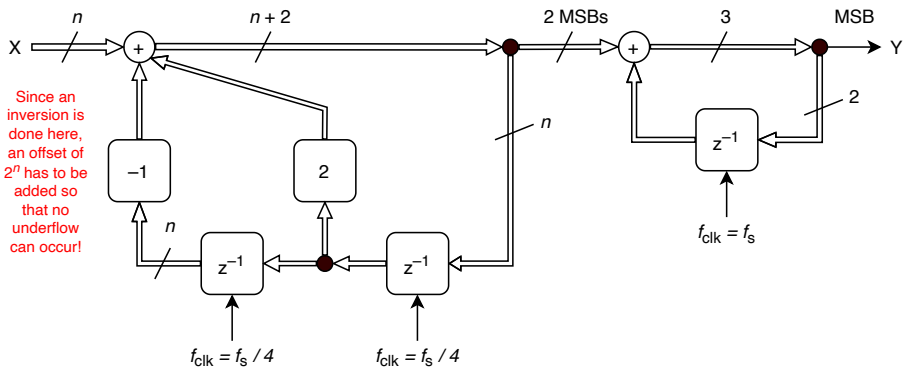

Figure 12: Delta-sigma error-feedback modulator of second order.

- https://www.chipverify.com/verilog/verilog-tutorial
- https://www.asic-world.com/verilog/veritut.html
- http://classweb.ece.umd.edu/enee359a/verilog_tutorial.pdf

The digital design flow is based on `OpenLane`, where all related information can be found at https://github.com/The-OpenROAD-Project/OpenLane. Please read the `README.md` file, as it explains how to handle the flow, the commands, and the file structure, and also gives a high-level overview of the various steps that are implemented when the flow is run.

For digital simulation of the `Verilog` code, we are using `Icarus Verilog`. Before running an actual simulation it is helpful to check the syntax, this can be done by linting the code. Here are two possibilities, the second tool provides a deeper checking of the code and can provide hints about potential pitfalls or bugs implemented in the `Verilog` code. The following script can be used to run linting (run the script w/o parameters to get a help screen):

```
> iic-vlint.sh <file.v>
```

The simulation can be started with the following commands, using the default filename of `a.out` for the compiled virtual machine (`audiodac.v` contains the implementation, the file `audiodac_tb.v` the testbench):

```
> iverilog -g2005 audiodac.v audiodac_tb.v
> ./a.out
```

7.1 Quick-start guide and tutorial

We have prepared a simple example of a binary counter in `Verilog`, and also show how to set up the environment. Please look at the file located at doc/iic_osic_verilog_tutorial.pdf.

For a good overview of the `Verilog` syntax please refer to doc/Verilog_Cheat_Sheet.pdf.

8 Exemplary implementation and verification scripts

8.1 Matlab/Octave code for architecture development

File `arch/dac_design.m`:

```
disp('-----------------------------------------');
disp('DAC Analyzer (c) 2021 Harald Pretl, IIC, JKU');
disp('-----------------------------------------');
disp('');
pkg load signal; % for `rms`, 'interp` and `decimate`
clear;

% name of audio tracks
% --------------------
audio_file_in  = 'testaudio.wav';
audio_file_out = 'testaudio_out.wav';

% parameters
% ----------
n = 16;
% no of bits; we represent data as unsigned integer
osr = 128;
% oversampling ratio of delta-sigma
interp_method = 1;
% 0 = zero insertion
% 1 = zero-order hold
% 2 = first-order hold [not implemented]
% 3 = interpolation with sinc
sd_type = 2;
% 1 = first-order modulator
% 2 = second-order modulator
use_sine = false;
% select if sine or audio used for testing

% read test audio file, is signed .wav (+1...-1)
% ----------------------------------------------
if use_sine == false
  disp(strcat('Step 1: Read audio file, filename=',audio_file_in));
  [track_in, fs_audio] = audioread(audio_file_in);
  track_in = track_in';
else
  disp('Step 1: Generate 441Hz sine tone with 60% fullscale');
  fs_audio = 44100;
  t_sin = (0:fs_audio-1)/fs_audio;
    track_in = 0.6 * sin(2*pi * 441 * t_sin);
  end % if

disp(strcat('Info: fs=',num2str(fs_audio),'Hz'))
% remove dc
track_in = track_in - mean(track_in);
% scale track (signed float to unsigned int by adding offset)
u = round((track_in+1)*2^(n-1));

% derived parameters
fullscale = 2^n;
fs_out = fs_audio * osr; % Hz

% upsample by osr, insertion of zeros
% -----------------------------------
if (interp_method == 0)
  % interpolation by inserting zeros
  u_samples = length(u);
  u_up = zeros(1,u_samples*osr);
  u_up(osr*(1:u_samples)) = u;
elseif (interp_method == 1)
  % interpolation by zero-order hold
  u_up = repelem(u,osr);
elseif (interp_method == 2)
  % interpolation by first-order hold
  % FIXME
elseif (interp_method == 3)
  % interpolation by sinc
  u_up = interp(u,osr);
end % if

% delta-sigma core
% ----------------
no_samples = length(u_up);
progress_step=round(no_samples/10);
reg1 = 0; reg2 = 0; reg3 = 0; c1 = 0;
out_sd = zeros(1,no_samples);

disp('Step 2: Perform delta-sigma processing...');

if (sd_type == 1) % first order modulator
  for i=1:no_samples
    % print a progress bar
    if (rem(i,progress_step) == 0)
      disp(strcat('  ',num2str(round(i/no_samples*100)),'%'));
    end % if

    out = u_up(i) + reg1; % 17b UINT
    reg1 = mod(out,fullscale); % 16b UINT; cut off MSB and store
    out_sd(i) = fix(out / fullscale); % 1b; MSB is output bit
  end % for
```

```
end % if

if (sd_type == 2) % second order modulator
  for i=1:no_samples
    % print a progress bar
    if (rem(i,progress_step) == 0)
      disp(strcat('  ',num2str(round(i/no_samples*100)),'%'));
    end % if

    if (mod(i,4) == 0) ; % first modulator runs on fs/4
      out1 = u_up(i) + 2*reg1 + (fullscale - reg2); % 18b UINT
      reg2 = reg1; % 16b UINT; reg2 = reg1 * z^-1
      reg1 = mod(out1,fullscale); % 16b UINT; cut off MSB/MSB-1 and store
      c1 = fix(out1 / fullscale); % 2b UINT; MSB/MSB-1 is output bits
    end % if
    out2 = c1 + reg3; % 3b UINT
    reg3 = mod(out2,4); % 2b UINT; cut off MSB and store
    out_sd(i) = fix(out2 / 4); % 1b; MSB is output bit
  end % for
end % if

disp('...done');

% decimate audio for playback, and scale according to input
% ---------------------------------------------------------
% we use a halfband filter approach to decimate
filt1 = fir1(osr, 1/osr);
filt2 = fir1(osr/4, 4/osr);
temp1 = fftfilt(filt1, out_sd);
temp2 = temp1(1:2:length(temp1));
temp3 = fftfilt(filt1, temp2);
temp4 = temp3(1:2:length(temp3));
temp5 = fftfilt(filt2, temp4);
out_sd_dec = temp5(1:(osr/4):length(temp5));
% now get back to audio track format
track_out = (out_sd_dec - mean(out_sd_dec)); % remove dc
track_out = track_out * rms(track_in)/rms(track_out); % balance looudness
track_out = min(0.999,track_out); % clip max
track_out = max(-0.999,track_out); % clip min
track_out = track_out * rms(track_in)/rms(track_out); % balance after clip

% write resulting audio track
% ---------------------------
disp(strcat('Step 3: Write audio file, filename=',audio_file_out));
audiowrite(audio_file_out,track_out',fs_audio);

% calculate SNR
% -------------
track_out(1:length(track_out)-1) = track_out(2:length(track_out));
sqnr_sd = 20*log10(rms(track_out)/rms(track_in-track_out));
disp(strcat('Info: SQNR=',num2str(sqnr_sd),'dB'));

% byebye
disp('');
disp('Done, bye!');
```

8.2 `Verilog` code for delta-sigma modulator

File `dig/rtl/audiodac.v`:

```
/*
 * AUDIODAC -- 16b Delta-Sigma Modulator with Single-Bit Output
 *
 * (c) 2021-2022 Harald Pretl (harald.pretl@jku.at)
 * Johannes Kepler University Linz, Institute for Integrated Circuits
 *
 * This delta-sigma modulator includes an input-side FIFO for easy connection
 * to a host system, volume control in 6dB steps, and configurable OSR
 * (32/64/128/256). The modulator can be configured to first or second order.
 *
 * The clock for the modulator is required to be clk = audio_sample_rate * OSR
 * e.g. for 44.1kHz and OSR=128 then clk=5.6448MHz
 *
 * The audio input data (fifo_i) must be 16b signed INT!
 *
 * In order to write into the FIFO:
 *   (1) assert the data at fifo_i[15:0] (with fifo_rdy_i=0)
 *   (2) signal that data is ready with a transition of fifo_rdy_i=0->1
 *   (3) wait that data is accepted by waiting for the transition fifo_ack_o=0->1
 *   (4) de-assert fifo_rdy_i=1->0 (this is followed by fifo_ack_o=1->0)
 *
 * Data is only accepted if the FIFO is not full (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 (fifo_full_o=0->1)
 *   (3) exit the ISR
 *
 * Ideas for future extension:
 *   (1) Implement a higher-order (e.g. 5th order) modulator.
 *   (2) Potentially substitute register-based FIFO by a dual-port RAM.
 */

`default_nettype none
`ifndef __AUDIODAC__
```

```
`define __AUDIODAC__

`ifdef SIM_PYTHON
    `include "../dig/rtl/iic_dsmod.v"
    `include "../dig/rtl/iic_fifo.v"
    `include "../dig/rtl/iic_sinegen.v"
 `else
    `include "iic_dsmod.v"
    `include "iic_fifo.v"
    `include "iic_sinegen.v"
`endif

module audiodac (
    // FIFO interface
    input       [15:0]  fifo_i,     // data is signed INT
    input               fifo_rdy_i,
    output wire         fifo_ack_o,
    output wire         fifo_full_o,
    output wire         fifo_empty_o,
    // housekeeping
    input               rst_n_i,
    input               clk_i,
    // configuration bits
    input               mode_i,     // 0 = 1st, 1 = 2nd order SD-mod
    input       [3:0]   volume_i,   // 0 = 0dB, 1 = -6dB, 2 = -12dB, ..., 15 = off
    input       [1:0]   osr_i,      // 0 = 32; 1 = 64, 2 = 128, 3 = 256
    // DS-modulator outputs
    output wire         ds_o,       // single-bit SD-modulator output
    output wire         ds_n_o,     //  plus the complementary output
    // test modes
    input               tst_fifo_loop_i,
    input               tst_sinegen_en_i,
    input       [4:0]   tst_sinegen_step_i
);

    // optional parameters for sine generator
    parameter SINE_GENERATOR_AMPL = 0.5;
    parameter SINE_GENERATOR_LUT_SIZE = 6;

    wire        [15:0]  dsmod_data_i, audio, audio_fifo, audio_sinegen;
    wire                audio_rd;
    assign              audio = tst_sinegen_en_i ? audio_sinegen : audio_fifo;
    // change 16-bit signed audio data to unsigned data
    assign              dsmod_data_i = audio + {1'b1,{15{1'b0}}};

    iic_fifo
        fifo0 (
            .fifo_indata_i(fifo_i),
            .fifo_indata_rdy_i(fifo_rdy_i),
            .fifo_indata_ack_o(fifo_ack_o),
            .fifo_full_o(fifo_full_o),
            .fifo_empty_o(fifo_empty_o),
            .fifo_outdata_o(audio_fifo),
            .fifo_outdata_rd_i(audio_rd),
            .rst_n_i(rst_n_i),
            .clk_i(clk_i),
            .tst_fifo_loop_i(tst_fifo_loop_i)
        );

    iic_dsmod
        dsmod0 (
            .data_i(dsmod_data_i),
            .data_rd_o(audio_rd),
            .ds_o(ds_o),
            .ds_n_o(ds_n_o),
            .rst_n_i(rst_n_i),
            .clk_i(clk_i),
            .mode_i(mode_i),
            .scale_i(volume_i),
            .osr_i(osr_i)
        );

    iic_sinegen
        // this module uses parameters, if your implementation does not use params
        // then delete the following line:
        #(16, SINE_GENERATOR_LUT_SIZE, SINE_GENERATOR_AMPL)
        sinegen0 (
            .data_o(audio_sinegen),
            .data_rd_i(audio_rd),
            .rst_n_i(rst_n_i),
            .clk_i(clk_i),
            .tst_sinegen_en_i(tst_sinegen_en_i),
            .tst_sinegen_step_i(tst_sinegen_step_i)
        );

endmodule // audiodac

`endif
`default_nettype wire
```

File `dig/rtl/audiodac_dsmod_template.v`:

```
/*
 * IIC_DSMOD -- Delta-Sigma Modulator (1st/2nd Order) with Single-Bit Output
 *
 * (c) 2021-2022 Harald Pretl (harald.pretl@jku.at)
 * Johannes Kepler University Linz, Institute for Integrated Circuits
```

```
 *
 * Ports:
 *      data_i          ... input data (must be UINT, default 16b)
 *      data_rd_o       ... fetch next input data word (data feed-in driven
 *                          by DS-modulator)
 *      ds_o, ds_n_o    ... output bitstream (complimentary outputs)
 *
 *      rst_n_i         ... reset (low active)
 *      clk_i           ... clock of ds-modulator (must be OSR * input_data_rate),
 *                          output bitstream rate = OSR * input_data_rate
 *
 *      mode_i          ... select order of modulator (0 = 1st, 1 = 2nd [optional])
 *      scale_i         ... scaling (attenuation) of input data in -6dB steps
 *                          (0 = 0dB, 1 = -6dB, 2 = -12dB, ..., 15 = off) [optional]
 *      osr_i           ... oversampling ratio (OSR), 0=32/1=64/2=128/3=256 is supported
 */

`default_nettype none
`ifndef __IIC_DSMOD__
`define __IIC_DSMOD__

module iic_dsmod (
    input       [15:0]      data_i,
    output wire             data_rd_o,
    output reg              ds_o,
    output wire             ds_n_o,

    input                   rst_n_i,
    input                   clk_i,

    input                   mode_i,
    input       [3:0]       scale_i,
    input       [1:0]       osr_i
);

    // Your code goes here!

endmodule // iic_dsmod

`endif
`default_nettype wire
```

File `dig/rtl/audiodac_fifo.v`:

```
/*
 * IIC_FIFO -- Configurable FIFO based on ring-buffer
 *
 * (c) 2021-2022 Harald Pretl (harald.pretl@jku.at)
 * Johannes Kepler University Linz, Institute for Integrated Circuits
 *
 * Ports:
 *      fifo_indata_i       ... input data
 *      fifo_indata_rdy_i   ... indicates that data is ready from source
 *      fifo_indata_ack_o   ... signales to source that datum has been taken into FIFO
 *
 *      fifo_full_o         ... indicates a full FIFO, no further datum will be
 *                              acknowledged into FIFO
 *      fifo_empty_o        ... indicates an empty FIFO, if datum is read from an empty
 *                              FIFO then the last datum will be output
 *
 *      fifo_outdata_o      ... output data
 *      fifo_outdata_rd_i   ... the next output datum is selected (if FIFO is not empty)
 *
 *      rst_n_i             ... reset (low active)
 *      clk_i               ... clock of FIFO
 *
 *      tst_fifo_loop_i     ... enables a test mode where the data in the FIFO is looped
 *                              at the output [optional]
 */

`default_nettype none
`ifndef __IIC_FIFO__
`define __IIC_FIFO__

module iic_fifo (
    input       [15:0]  fifo_indata_i,
    input               fifo_indata_rdy_i,
    // note that fifo_indata_i and fifo_indata_rdy_i could originate from
    // an asynchronous clk domain, so potentially needs to be synchronized
    // use FIFO_ASYNC=1 to select the proper behaviour
    output reg          fifo_indata_ack_o,
    output wire         fifo_full_o,
    output wire         fifo_empty_o,
    output wire [15:0]  fifo_outdata_o,
    input               fifo_outdata_rd_i,
    input               rst_n_i,
    input               clk_i,
    input               tst_fifo_loop_i
);

   // Your code goes here!

endmodule // iic_fifo

`endif
`default_nettype wire
```

File `dig/rtl/audiodac_sinegen.v`:

```
/*
 * IIC_SINEGEN -- Simple Sine Generator based on a harcoded 16b LUT
 *
 * (c) 2021-2022 Harald Pretl and Jakob Ratschenberger (harald.pretl@jku.at)
 * Johannes Kepler University Linz, Institute for Integrated Circuits
 *
 * Ports:
 *      data_o              ... output data of sine generator (BW)
 *      data_rd_i           ... advances the generator to the next output value,
 *                              controlled by the step size
 *
 *      rst_n_i             ... reset (low active)
 *      clk_i               ... clock of sine generator (pos. edge)
 *
 *      tst_sinegen_en_i    ... enables the sine generator; when disabled then 0 is
 *                              output, and the generator is turned off
 *      tst_sinegen_step_i  ... controls the step size per read (1 = use every ROM
 *                              value, 2 = use every 2nd ROM entry, etc); the higher the
 *                              step value, the faster the output sine
 */

`default_nettype none
`ifndef __IIC_SINEGEN__
`define __IIC_SINEGEN__

module iic_sinegen (
    output wire signed [15:0]   data_o,
    input                       data_rd_i,
    input                       rst_n_i,
    input                       clk_i,

    input                       tst_sinegen_en_i,
    input [3:0]                 tst_sinegen_step_i
);

    // Your code goes here!

endmodule // iic_sinegen

`endif
`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`:

```
"""
    AUDIODAC_TB

    (c) 2022 Jakob Ratschenberger
    Johannes Kepler University Linz, Institute for Integrated Circuits

    Python script to controll and perform the simulation of the AUDIODAC.

    Implemented functionality:
        - Setting the AUDIODAC configurations (OSR, MODE, VOLUMNE)
        - Setting the AUDIODAC input data
            -- Four sets are available:
                + Filtered White Gaussian Noise
                + Sine with 10kHz
                + All zeros
                + Sound of an bell
        - Data visualisation
            -- Plotting the input data in time and frequency domain
            -- Plotting the output data in frequency domain
        - Generating .wav of the AUDIODAC output for the sound input
            -- 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 AudioDACSimVals as simvals
32 import AudioDACSimResults 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_OSR: (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_OSR = 0
47 if OSR == 64:
48     SIM_OSR = 1
49 elif OSR == 128:
50     SIM_OSR = 2
51 elif OSR == 256:
52     SIM_OSR = 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 volumne
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 (white Gaussian noise)
70          (2) Sine (10kHz)
71          (3) Zeros
72          (4) Sound (bell)
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     os.system("rm -f AudioDAC.wav")
98     sound_sample_rate, test_data = wavfile.read("Bell.wav")
99     nSamples = len(test_data)
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'$f\,/\,Hz$')
```

```
ax.set_ylabel(r'$\mid \mathrm{fft}\left(x_{Test}[n]\right) \mid$ ')
ax.grid(True)
plt.draw()

# Setup the simulation values
mySimVals = simvals.AudioDACSimVals(SIM_MODE, SIM_OSR, SIM_VOLUME, test_data)

# Generate the necessary files for the sim
mySimVals.genSimFiles()

# Run the verilog simulation (remove old compiled TB and results file first)
os.system("rm -f AUDIODAC_PY_TB")
os.system("rm -f verilog_bin_out.txt")
os.system("iverilog -g 2005 -o AUDIODAC_PY_TB -c file_list.txt")
os.system("vvp AUDIODAC_PY_TB")

# Set the file name of the simulation result
mySimResults = simresults.AudioDACSimResults("verilog_bin_out.txt")

# Read the simulation result
data = mySimResults.getSIM_Result()

data = (data)*2-1       # Merge the poitive and negative AUDIODAC output

data = data*2**(BW-1)   # Scale the data to input range

# Calc. the spectrum of the AUDIODAC output stream
freq_data = np.fft.fft(data, norm="forward")
f = np.fft.fftfreq(freq_data.size,d=1/(fs*OSR))

# Plot the spectrum of the output data
fig, ax = plt.subplots()
fig.suptitle(r'Spectrum of the output data')
x = np.fft.fftshift(f)
y = np.abs(np.fft.fftshift(freq_data))
ax.plot(x,y, 'b')
ax.set_xlabel(r'$f\,/\,Hz$')
ax.set_ylabel(r'$\mid \mathrm{fft}\left(x_{out}[n]\right) \mid$')
ax.grid(True)
plt.draw()

# Get the output spectrum in the range 0 to fs/2
indx_fs = round(len(freq_data)/(OSR*2))
freq_data2 = freq_data[0:indx_fs-1]
f = f[0:indx_fs-1]

# Plot the output spectrum in the range 0 to fs/2
fig, ax = plt.subplots()
fig.suptitle(r'Zoomed spectrum of the output data')
x = f
y = np.abs(freq_data2)
ax.plot(x,y,'b')
ax.set_xlabel(r'$f\,/\,Hz$')
ax.set_ylabel(r'$\mid \mathrm{fft}\left(x_{out}[n]\right) \mid$')
ax.grid(True)
plt.draw()

# If the test set was the sound file
if TestSet == 4:
    # Resample the AUDIODAC output stream to fs by using Fourier method
    nSamples = round(len(data)/OSR)
    data = signal.resample(data,int(nSamples))

    # Write the output to a .wav file
    wavfile.write("AudioDAC.wav",fs, data.astype(np.int16))

    # Plot the waveform
    fig, ax = plt.subplots()
    fig.suptitle(r'Waveform of the output data')
    ax.plot(data)
    ax.grid(True)
    plt.draw()

# Show all figures at the end
plt.show()
```

File `dig/rtl/audiodac_python_tb.m`:

```
/*
 * AUDIODAC TESTBENCH -- 16b Delta-Sigma Modulator with Single-Bit Output
 *
 * The input data gets provided by the use of a Python script.
 *
 * (c) 2021-2022 Harald Pretl (harald.pretl@jku.at)
 * Johannes Kepler University Linz, Institute for Integrated Circuits
 *
 * This is the testbench for <audiodac.v>
 */

`timescale 10ns / 1ns

`ifndef SIM_PYTHON
`define SIM_PYTHON
`endif

`include "../dig/rtl/audiodac_simParam.v"

```

```
module audiodac_python_tb;

    localparam SIM_MODE = `SIM_MODE;
    localparam SIM_OSR = `SIM_OSR;
    localparam SIM_VOLUME = `SIM_VOLUME;

    localparam DATA_SAMPLES = `SIM_DATA_SAMPLES;

    // large memory to hold the audio
    reg signed [15:0]   DATA_IN[0:DATA_SAMPLES-1];
    // output results written to file
    reg                 DATA_OUT[0:(DATA_SAMPLES*(32*2^SIM_OSR)-1)];

    integer             DATA_IN_CTR = 0;
    integer             DATA_OUT_CTR = 0;

    // configuration bits
    reg                 MODE = SIM_MODE;
    reg [1:0]           OSR = SIM_OSR;
    reg [3:0]           VOLUME = SIM_VOLUME;

    // inputs
    reg                 RESET_N = 1'b0;
    reg                 CLK = 1'b0;
    reg                 TST_FIFO_LOOP = 1'b0;
    reg                 TST_SINEGEN_EN = 1'b0;
    reg [4:0]           TST_SINEGEN_STEP = 5'd2;

    // outputs
    wire                FIFO_FULL, FIFO_EMPTY, FIFO_ACK, DS_OUT, DS_OUT_N;

    // instantiate DUT
    audiodac dac (
        .fifo_i(DATA),
        .fifo_rdy_i(DATA_RDY),
        .fifo_ack_o(FIFO_ACK),
        .fifo_full_o(FIFO_FULL),
        .fifo_empty_o(FIFO_EMPTY),
        .rst_n_i(RESET_N),
        .clk_i(CLK),
        .mode_i(MODE),
        .volume_i(VOLUME),
        .osr_i(OSR),
        .ds_o(DS_OUT),
        .ds_n_o(DS_OUT_N),
        .tst_fifo_loop_i(TST_FIFO_LOOP),
        .tst_sinegen_en_i(TST_SINEGEN_EN),
        .tst_sinegen_step_i(TST_SINEGEN_STEP)
    );

    // make a clock
    always #1 CLK = ~CLK;

    // handle FIFO input data
    reg signed [15:0]   DATA = 16'b0;
    reg                 DATA_RDY = 1'b0;
    // flag to signal a wait for the next write burst
    reg                 WAIT_FOR_EMPTY = 1'b0;

    always @(negedge CLK) begin
        // handling of simulation input and output data is done at falling CLK edge
        // IP block is clocked by rising edge

        // provide input data
        if (!DATA_RDY && !FIFO_FULL && !FIFO_ACK && !WAIT_FOR_EMPTY && RESET_N) begin

            DATA <= DATA_IN[DATA_IN_CTR];
            DATA_IN_CTR = DATA_IN_CTR + 1;

            // signal to FIFO that data is ready
            DATA_RDY <= 1'b1;

            // no more input data left? write result and exit
            if (DATA_IN_CTR == DATA_SAMPLES) begin
                $writememh("verilog_bin_out.txt", DATA_OUT);
                $finish;
            end
        end

        // de-assert data_rdy when data transfer ack'd by FIFO
        if (FIFO_ACK) DATA_RDY <= 1'b0;

        // The FIFO data write-in is done in a bursty nature, like a
        // host system would likely do. When the FIFO is empty we write
        // until the FIFO is full, then we wait for the FIFO to get
        // empty again--then we do another bursty data transfer. This
        // is meant to put less burden on the host system, so just an
        // interrupt service routine is required, no constant polling
        // of the data_rdy line, instead fifo_empty can trigger the ISR.

        // stall data write-in when FIFO is already full
        if (FIFO_FULL) WAIT_FOR_EMPTY <= 1'b1;

        // if FIFO is empty re-engage with data write-in
        if (FIFO_EMPTY) WAIT_FOR_EMPTY <= 1'b0;

        // store simulation result into memory for later dump
        if (RESET_N) begin
            DATA_OUT[DATA_OUT_CTR] <= DS_OUT;
```

```
119             DATA_OUT_CTR = DATA_OUT_CTR + 1;
120         end
121     end
122
123     // here is all the initiliaztion work for the simulation
124     initial begin
125         // print out simulation state
126         $display("-----------------------------");
127         $display("SIM_MODE   = ", SIM_MODE);
128         $display("SIM_OSR    = ", SIM_OSR);
129         $display("SIM_VOLUME = ", SIM_VOLUME);
130         $display("-----------------------------");
131
132         $readmemh("audiodac_test_data.txt", DATA_IN);
133
134         `ifndef SIM_NO_VCD
135             $dumpfile("audiodac_tb.vcd");
136             $dumpvars;
137         `endif
138
139         // de-assert reset
140         #5 RESET_N = 1'b1;
141     end
142
143 endmodule // audiodac_python_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.