



AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

Komponenter til lydbehandling i en FPGA

ETDSPC Projekt

Rune Salberg-Bak (08935), Kim Bjerge (20097553)

Q1 2012

Indhold

| | | |
|-------|--|----|
| 1 | Indledning | 3 |
| 2 | Problemformulering | 3 |
| 3 | Udviklingsstrategi | 5 |
| 4 | Arkitektur | 6 |
| 5 | Komponent beskrivelser og verifikation | 12 |
| 5.1 | ModelSim Testbench | 12 |
| 5.2 | FIR filter | 15 |
| 5.3 | LMS filter | 19 |
| 5.3.1 | FPGA Implementering | 21 |
| 5.4 | Delay | 27 |
| 5.5 | Sigma delta konverter | 30 |
| 6 | SoPC systemer | 35 |
| 6.1 | SoPC - LMS filter med audio delay | 35 |
| 6.2 | SoPC - Sigma delta konverter | 38 |
| 7 | Resultater og diskussion | 40 |
| 8 | Konklusion | 42 |
| 9 | Appendix A - Modeller | 43 |
| 9.1 | MATLAB – LMS filter | 43 |
| 9.2 | C-Kode LMS filter | 45 |
| 9.3 | MATLAB – Sigma Delta Converter | 47 |
| 10 | Appendix B – VHDL og SOPC oversigt | 51 |
| 10.1 | VHDL kode med test benches | 51 |
| 10.2 | SOPC projekt arkiver | 52 |
| 11 | Referencer | 52 |

1 Indledning

Denne rapport har til formål at demonstrere opnåelse af læringsmålene fra kurset "Design of Systems on Programmable Chips" (ETDSPC). Vi har valgt at arbejde med udvikling af komponenter til lydbehandling i en FPGA. En FPGA løsning til dette problem er velegnet, som alternativ til brug af en DSP. Vi får en klar adskillelse mellem real-time lydbehandlingen og den applikationssoftware, som kan benyttes til andre af systemets funktioner. Det kunne være betjening, kommunikation eller beregning af nye filter koefficienter til eksempelvis ændring af egenskaber for lydbehandlingen. FPGA løsningen kan optimeres så meget skrappe timings krav kan i møde kommes uden at belaste softcore processoren. Med customized instruktioner til softcore processoren, er det også muligt at accelerere afvikling på processoren. Løsningen kunne tænkes anvend til professionelle audio produkter, hvor man arbejder med en samplingsfrekvens op til 192 KHz, med komplicerede udregning af nye filterkoefficienter. Det kunne også være til ultralydsskanning af fostre hvor samme høje samplingsfrekvenser er påkrævet. Alle audio komponenter implementeret i dette projekt kunne benyttes i denne type applikationer. Der kræves ikke ekstra arbejde med potering til en specifik arkitektur, hvis komponenterne skulle benyttes i en anden FPGA platform end DE2 boardet fra Altera. I dette projekt har vi vagt at begrænse os til et stereo audio format på 48 KHz med en opløsning på 24 bits.

Vi benytter en simpel version af Avalon Streaming Interfacet (ST Bus) [6], med mulighed for ændring af funktionalitet fra software afviklet på Altera's Nios II processorer. Dette valg dækker mange af de emner, som vi har arbejdet med i kurset og sigter mod opfyldelse af læringsmålene for kurset. I konklusionen har vi opsummeret hvilke læringsmål, der er demonstreret opfyldt i dette projekt.

FIR Filter design, LMS filter og audio delay komponenterne med tilhørende test benches og SoPC projekt er implementeret og beskrevet i denne rapport af Kim Bjerge. Rune Salber-Bak har implementeret Sigma Delta Converteren med tilhørende ST bus multiplekser, test benches samt SoPC projekt med et analogt aktivt filter.

Rune's arbejde er ikke beskrevet i rapporten, kontakten til ham er ikke etableret igen efter projekt arbejdsgang fredag d. 9. marts. Kim har forgæves forsøgt af få kontakt mange gange via. mails og telefon men uden held, derfor er væsentlige kapitler vedrørende sigma delta konverterne arbejde udeladt af denne rapport. Den 9. marts demonstrerede Rune en funktionsdygtig stereo version af sigma delta konverter med aktivt analogt filter, som heller ikke har været muligt at dokumentere i rapporten.

2 Problemformulering

Dette projekt har som mål, at udvikle forskellige audio komponenter til behandling af lyd i et FPGA design. Udgangspunktet er et DE2 board fra Altera. DE2 Boardet har et codec til håndtering af stereo lyd (LINE IN/OUT) som overføres på I2S format mellem codec og FPGA. I kurset ETDSPC har vi haft øvelser med implementering af komponenter for konvertering af I2S til Avalon Streaming Bus (ST). Dette projekt vil implementere forskellige lydbehandlings komponenter, der kan benyttes i et Altera SOPC design.

Komponenter til lydbehandling i en FPGA

Komponenters opsætning skal kunne konfigureres med brug af VHDL generics. Ændring af komponenternes parameter skal kunne styres fra Nios II processoren. Hertil benyttes Avalon Memory-Mapped Interface (MM) [6]. Audio komponenterne udvikles og testes med simulering i ModelSim. Komponenter skal være udviklet så de i principippet kunne flyttes til en anden type FPGA som f.eks. Xilinx, med omskrivning af memory interface til processoren (MicroBlaze). Dette krævet, at vi ikke benytter de indbygget Altera komponent wizzards eller biblioteker men implementere vores egne komponenter med ST bus interface i VHDL. Komponenterne skal implementeres i VHDL og optimeres for et design med digital stereo lyd i 24 bits format og en samplings rate på 48 kHz.

De algoritmer vi har valgt at implementere tager udgangspunkt i kurset ETDSPC samt andre signalbehandlingskurser vi har fulgt på vores studie. Udgangspunktet er modeller af algoritmerne i MATLAB og/eller C-kode som vi har haft i øvelser eller projekter. Målet er implementering og optimering af disse algoritmer med brug af den teori vi haft i faget ETDSPC.

Nedenfor er listet de audio komponenter, vi har valgt at arbejde med:

1. LMS Filter

- Implementering af et adaptivt LMS filter, optimeret for minimering af FPGA area
 - Her er målet at fjerne brum eller støj fra et signal med en kendt støj kilde

2. Stereo Delay

- Implementering af en forsinkelse af lyden med de indbyggede FPGA ram blokke
 - Målet er at anvende Alteras FPGA ram blokke implementeret i VHDL

3. Sigma Delta Konverter

- Implementering af en sigma delta konverter
 - Her bruger vi digitale FPGA ben, hvor vi kan afspille stereo lyd efterfulgt af et aktivt analogt filter

4. Demonstration af prototyper

- Ovenstående komponenter demonstreres på et Altera DE2 board for SoPC designs med tilhørende test software

Nedenfor er listet de arbejdsopgaver vi har identificeret for projektet:

- Design og implementering af en test bench i ModelSim, der kan indlæse filer med audio samples generet af modellerne i MATLAB eller C-kode
- Udvikling af forskellige FIR filter typer (Direkte, Transposed og symmetrisk)
 - Implementeres i VHDL og testes i ModelSim
 - Skal senere anvendes af LMS filter og sigma delta konverter
- Test bench verifikation af algoritmer i forhold til referencer modeller

- Modeller i MATLAB eller C-kode genererer tekst filer med audio samples som sammenholdes med VHDL verifikation i ModelSim
- Verifikation af LMS filter, delay og sigma delta konverter
- SoPC design med udgangspunkt i kursets øvelser
 - Design med opsætning af audio Codec via. I2C og streaming af audio via. I2S
 - Inkludere nogle af fagets øvrige små øvelser (Custom instructions, 7-segment)
- Flere SoPC projekter med stereo line in/out - 48 kHz/24 bit
 - LMS filter med audio delay
 - Sigma delta konverter i stereo med aktivt analog filter
- Oversigt af udviklede audio komponenter (ST, MM kompatible) med information om ressource forbrug som f.eks. area (LE, Multipliers, Block RAM), latency, throughput
- Denne rapport med beskrivelse af arkitektur, implementering samt refleksioner over resultater og læring

3 Udviklingsstrategi

Designet af audio komponenterne tager udgangspunkt i signalbehandlingsteorien. Forskellige formler og algoritmer afprøves i en simuleret model på et højere abstraktionsniveau inden den egentlig implementering i software eller hardware. Målet med denne simulerede model er at undersøge om den ønskede algoritme kan løse en given opgave. I dette projekt kunne kravet f.eks. være, at designe en sigma delta konverter med et teoretisk signal/støjforhold på mindst 37 dB. Hvilken oversamplings-rate er bedst? Hvilken filterorden skal vælges? Hvad giver et 1. ordens eller 2. ordens noise-shaping kvantiseringsfilter som forbedring? Denne type spørgsmål kan bedst besvares med en model på et højre abstraktionsniveau som er muligt med modeller i MATLAB. Når en given algoritme er simuleret på dette niveau, er det næste step at omforme algoritmen til en given target implementering. Det kunne være en DSP eller FPGA platform. Den næste udfordring er at bestemme algoritmens regne nøjagtighed for de givne krav om oplosning herunder fixed-point format og input/output format. I dette tilfælde skal input samples med en oplosning på 24 bit. Ved implementering i fixed-point format kan algoritmen modelleres f.eks. i C-kode eller MATLAB. Algoritmens regne præcision afprøves inden implementering på den endelige platform. I dette projekt har vi haft en model af LMS algoritmen i MATLAB og fixed-point C-kode. For sigma delta konverteren er udgangspunktet en model i MATLAB. Se appendix A for flere detaljer om disse modeller.

I projektet har vi fokuseret på at implementere ovenstående modeller i VHDL med DE2 boardet om target platform. Strategien er, at implementere en version af algoritmen i VHDL, som først simuleres og aftestes med en ModelSim test bench. Simuleringen benytter input test data produceret af modellerne fra MATLAB eller C-kode, hvor output resultatet sammenlignes med den "gyldne" reference model. Fokus punkter for implementering i VHDL er emner som: interface til ST bussen, optimering i forhold til area, latency og throughput. Med en samplings rate på 48 KHz og med en 12 MHz clock frekvens på ST bussen, har vi masser af tid (clock cycler) til processering af audio data. Derfor har fokus været at minimere brugen af FPGA ressourcerne som f.eks. antallet af multiplikationer. ModelSim modellerne er verificeret i en funktionel simulering, hvor vi ikke har taget højde for gates og kombinatorisk forsinkelser. Når en algoritme er verificeret, har vi kompileret VHDL koden for komponenten i et Quartus projekt for at bestemme

forbruget af FPGA ressourcer herunder: Logiske Elementer (LE), Registeres (FlipFlops), Multipliers, RAM blokke og med timingsanalyse bestemme den maksimum clock frekvens (Fmax).

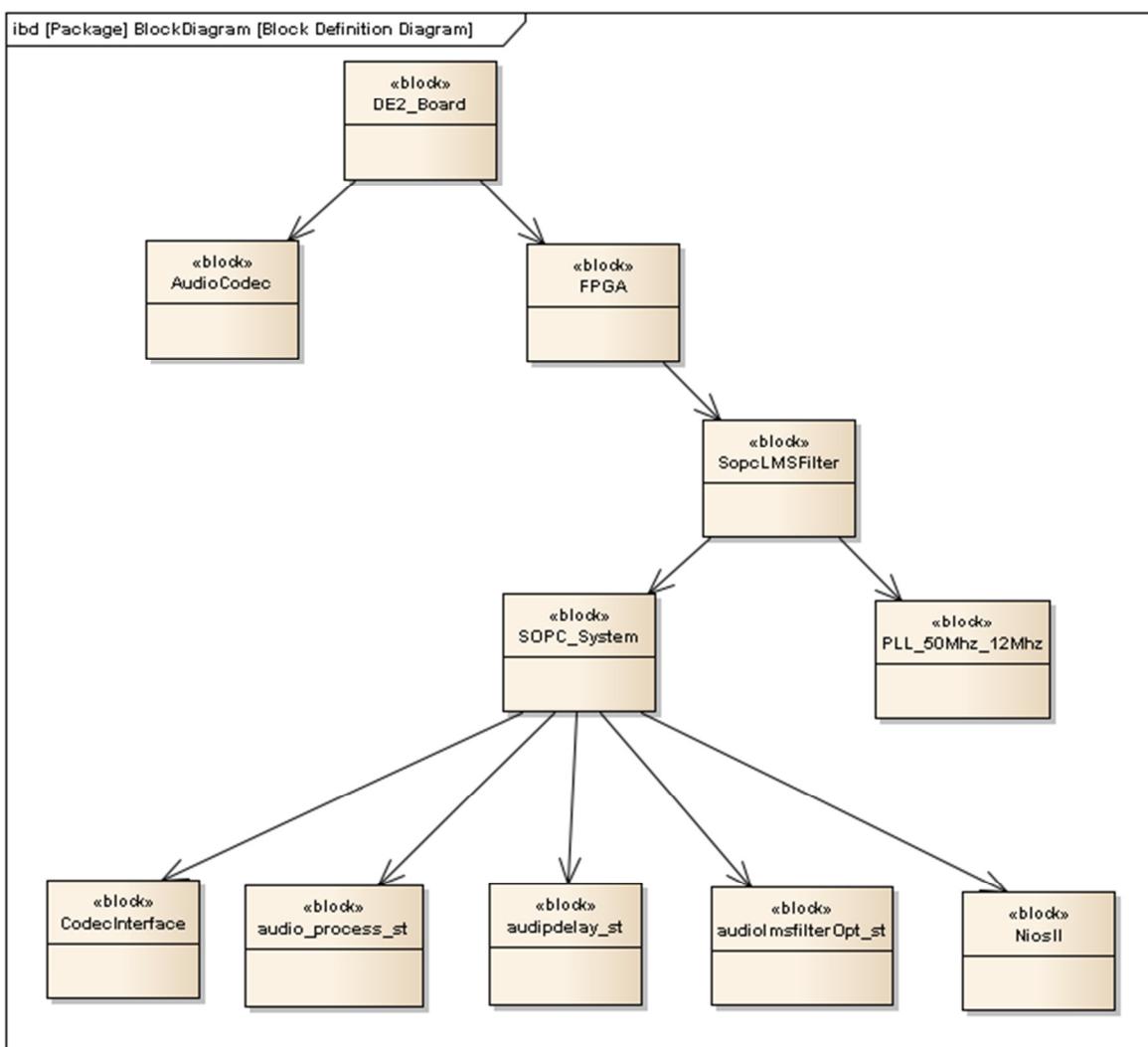
Der er udviklet flere forskellige SoPC projekter med Nios II processoren, hvor vi løbende har indsat versioner af komponenterne efter afprøvning i ModelSim. Det har vist sig at være en god strategi. Selv om komponenten er testet i ModelSim, er det ikke altid det virkede på DE2 boardet. Det kunne f.eks. være, hvis vi havde glemt at initialiserer et vigtigt signal. Således har vi step vist skiftet mellem at aftestet ændringer i ModelSim og efterfølgende i systemet på target. Med versions kontrol (SubVersion), har vi hele tiden haft en gammel fungerende version vi kunne sammenligne med hvis noget gik galt.

Vi startede med et SoPC design, hvor vi har benyttet et VHDL komponent udleveret i kurset, med konvertering fra I2S til ST format. Et separat I2C komponent foretager kommunikation og initialisering af codec via. softwaren på Nios II processoren. Dette første SoPC projekt har vi haft store vanskeligheder med. Selv med et simpelt design, hvor vi bare ruter lyden direkte fra ST-Source til ST-Sink, har vi ikke kunnet få til at virke stabilt. Problemet er at softwaren ikke kunne downloades. Der gives en check sum fejl efter download til SRAM memory på DE2 boardet. Efter mange timers forsøg ændrede vi strategi og benytter i de efterfølgende SoPC designs et færdig udviklet komponent af vores underviser, der indeholder en kombineret I2S til en speciel audio synkron parallel bus, som beskrives i det efterfølgende afsnit. Komponenten indeholder en direkte I2C kommunikation med opsætning af Codec fra FPGA hardwaren. Denne strategi er ikke så fleksibel som i det første design, men mere stabilt i forhold til software fejl. Efter en hardware reset vil Codec altid initialiseres korrekt (Master, 24 bit, I2S, 48 KHz). Dette SoPC projekt har været vores grundlag for implementering af et ST Bus interface og audio komponenterne. I det følgende afsnit, har vi beskrevet systemets arkitektur med MM og ST busser, komponenter for de to færdige SoPC systemer med henholdsvis sigma delta konverter og LMS filteret med stereo delay.

4 Arkitektur

Den overordnede arkitektur er beskrevet i nedenstående SysML diagrammer, med de komponenter som er væsentlige for vores projekt. DE2 boardet har monteret et audio codec fra Wolfson Microelectronics (WM8731), som det centrale komponent for udveksling af digital audio over I2S med FPGA'en. SoPC designet for LMS filteret (**SopcLMSFilter**) indeholder et SOPC_System genereret med Altera's SOPC Builder. De væsentlige komponenter er et **CodecInterface**, der indeholder opsætning af audio codec via. I2C protokollen. Komponenten **audio_process_st** udvikles med formål at konvertere et synkront audio interface fra **CodecInterface** til ST bussen. Komponenterne **audiodelay_st** og **audiolmsfilterOpt_st** skal sammen med software til Nios II processoren udvikles i projekt. Der er indsat en PLL til generering af et 50 MHz og 12 MHz clock, der sikre samme fase og dermed samme clock domæne.

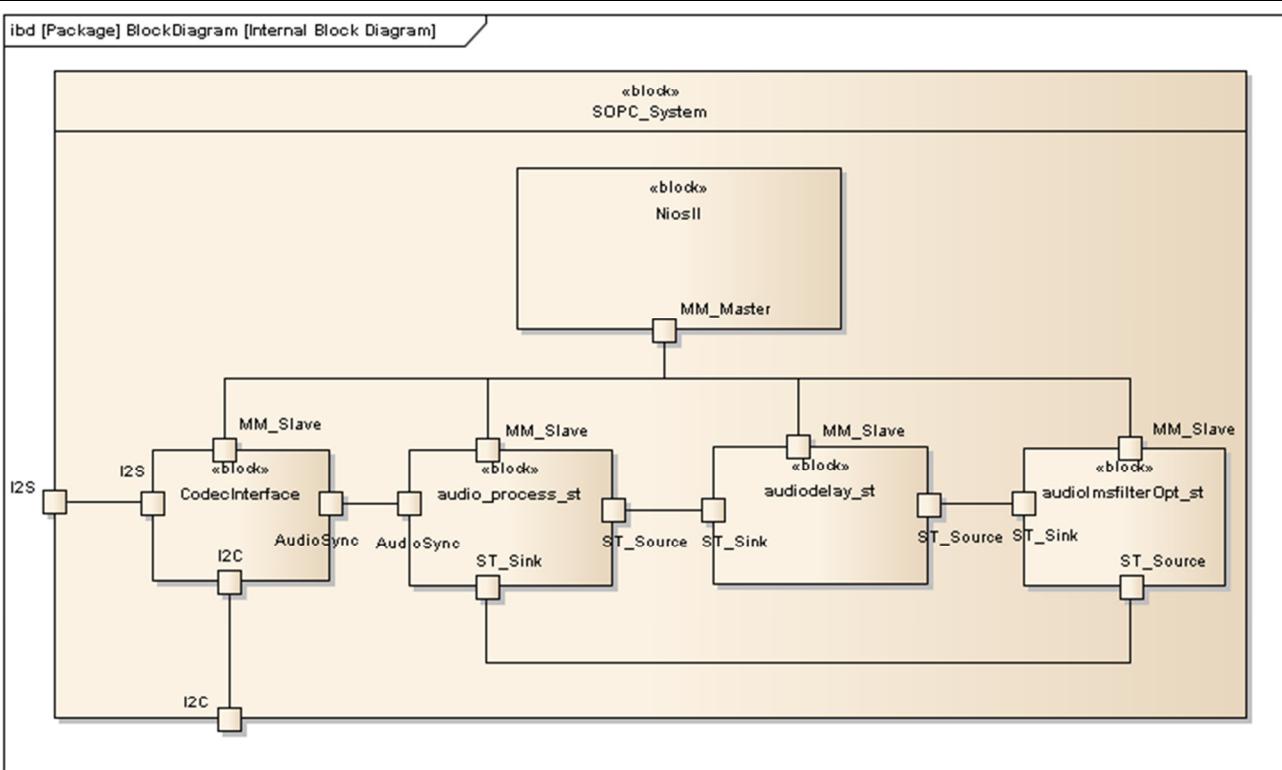
Komponenter til lydbehandling i en FPGA



Figur 1 SysML arkitektur diagram

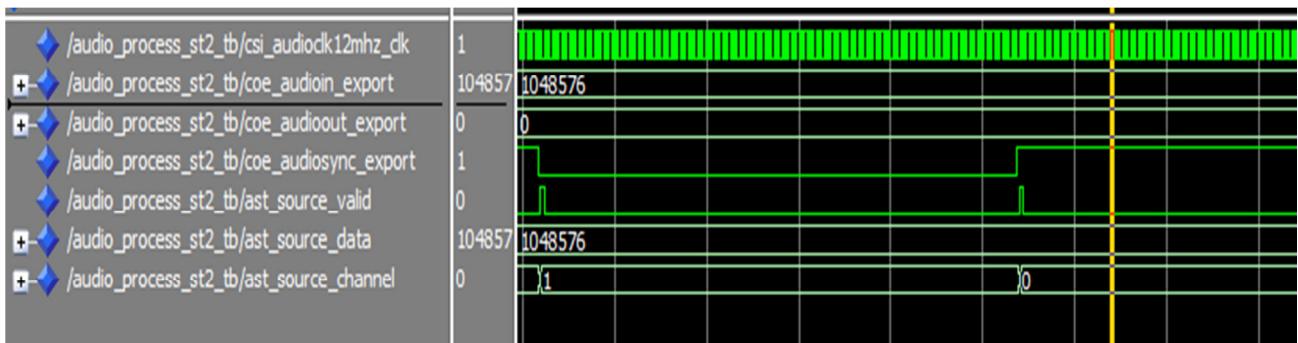
Nedenstående diagram viser, hvordan audio komponenterne er forbundet på henholdsvis Avalon Memory Management bussen (MM) og Streaming (ST) interfacet, hvor audio delay og LMS filteret er indsatt i serie sammen med konvertering til ST interfacet.

Komponenter til lydbehandling i en FPGA



Figur 2 SysML internt arkitektur diagram for SOPC Systemet til LMS filter og audio delay

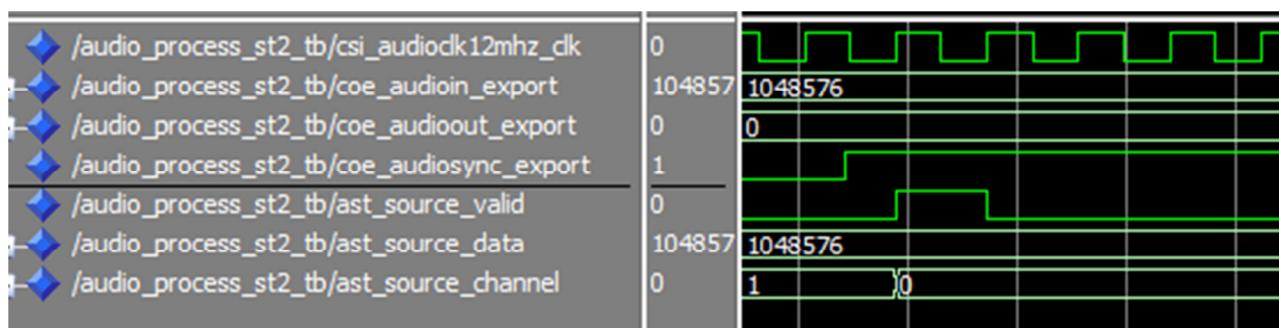
Audio sync interfacet, består af et signal, der for henholdsvis værdien 1 og 0 angiver venstre og højre audio kanal. Audio sendes på denne parallelle bus, som består af interface forbindelserne (AudioIn, AudioOut og AudioSync) se Figur 3.



Figur 3 Audio sync til ST bus interface

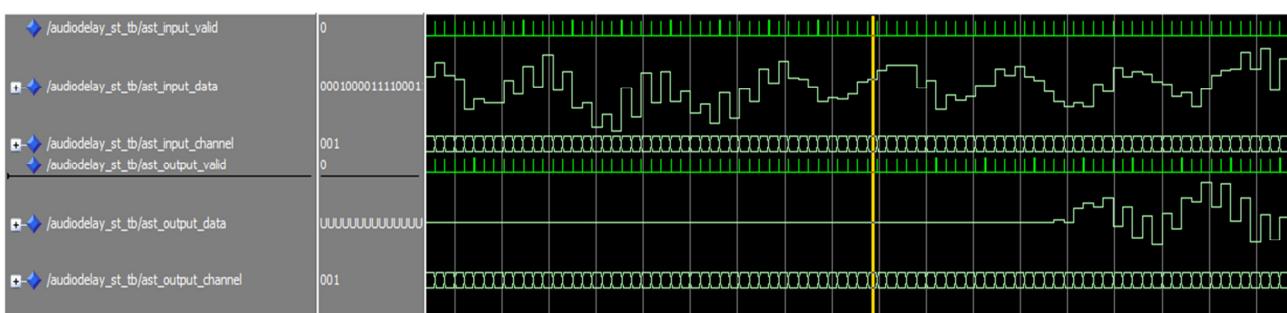
Vi har valgt kun at benytte signalerne: valid, data og channel fra Avalon's specifikation af ST interfacet. Channel angiver et nummer på den kanal, hvortil data er knyttet angivet med 3 bits. Hermed er en udvidelse muligt til f.eks. håndtering af surround, 5.1 lyd formatet med 6 audio kanaler. Valid holdes højt i en 12 MHz clock cycle. ST source sætter valid høj sammen med channel og data når 12 Mhz clock'en går høj. ST sink sampler data og channel nummer når 12 MHz klokken går lav som vist i Figur 4. ST ready signalet er udeladt for at spare logik, dette har dog senere vist at være en dårlig idé, hvis komponenterne skal benyttes sammen med nogle af Altera's Streaming komponenter.

Komponenter til lydbehandling i en FPGA



Figur 4 ST bus timing for valid signal

Nedenstående timings diagram er fra en simulering med ModelSim, der viser hvordan ST interfacet ser ud når et audio delay indsættes som komponent på ST bussen.



Figur 5 Audio delay indsæt på ST bussen

Vi har valg at vise hele VHDL koden for **audio_process_st**, som benyttes af alle komponenter, der er beskrevet i de følgende afsnit om komponenterne. Her vises, hvilke forbindelser der indgår i audio interfacet, ST og MM bussen. Med software over MM bussen kan lyden mutes for højre og venstre kanal.

```

entity audio_process_st is
  generic (audioWidth : natural := 24;
           chNrLeft      : std_logic_vector(2 downto 0) := "000";
           chNrRight     : std_logic_vector(2 downto 0) := "001");
  port (
    -- Audio Interface
    csi_AudioClk12MHz_clk      : in  std_logic;
    csi_AudioClk12MHz_reset_n   : in  std_logic;
    coe_AudioIn_export         : in  std_logic_vector(audioWidth-1 downto 0);
    coe_AudioOut_export        : out std_logic_vector(audioWidth-1 downto 0);
    coe_AudioSync_export       : in  std_logic;

    -- Avalon MM Bus
    csi_clockreset_clk          : in  std_logic;
    csi_clockreset_reset_n      : in  std_logic;
    avs_s1_write                 : in  std_logic;
    avs_s1_read                  : in  std_logic;
    avs_s1_chipselect            : in  std_logic;
    avs_s1_address                : in  std_logic_vector(7 downto 0);
    avs_s1_writedata             : in  std_logic_vector(15 downto 0);
    avs_s1_readdata              : out std_logic_vector(15 downto 0);

    -- ST Bus
    ast_source_valid             : out std_logic;
  
```

```
ast_source_data           : out    std_logic_vector(23 downto 0);
ast_source_channel        : out    std_logic_vector(2 downto 0);
ast_sink_valid            : in     std_logic;
ast_sink_data              : in     std_logic_vector(23 downto 0);
ast_sink_channel           : in     std_logic_vector(2 downto 0)

);

end audio_process_st;

architecture behaviour of audio_process_st is

constant CI_ADDR_START      : std_logic_vector(7 downto 0) := X"00";
constant CI_ADDR_STATUS      : std_logic_vector(7 downto 0) := X"40";
constant CI_UNMUTED          : std_logic                         := '0';

-- Internal signals
signal AudioSync_last       : std_logic;
signal mute_left              : std_logic;
signal mute_right             : std_logic;

signal left_sample            : std_logic_vector(audioWidth-1 downto 0);
signal right_sample           : std_logic_vector(audioWidth-1 downto 0);

signal valid_high             : std_logic := '0';

begin

-----  

-- purpose: Register with Avalon Bus interface
-- inputs : csi_clockreset_clk, csi_clockreset_reset_n, avalonbus
-----  

accessMem : process (csi_clockreset_clk, csi_clockreset_reset_n)
  variable wrData : std_logic_vector(avs_s1_writedata'high downto 0);
begin -- process accessMem

  if csi_clockreset_reset_n = '0' then -- asynchronous reset (active low)
    mute_left <= CI_UNMUTED;
    mute_right <= CI_UNMUTED;

  elsif csi_clockreset_clk'event and csi_clockreset_clk = '1' then

    if avs_s1_chipselect = '1' then
      if avs_s1_write = '1' then
        case avs_s1_address is
          when CI_ADDR_START =>
            mute_right <= avs_s1_writedata(0);
            mute_left <= avs_s1_writedata(1);
          when others  => null;
        end case;
      end if;

      if avs_s1_read = '1' then
        if avs_s1_address = CI_ADDR_START then
          avs_s1_readdata <= (0 => mute_right, 1 => mute_left, others => '0');
        else
          avs_s1_readdata <= (others => '0');
        end if;
      end if;
    end if;
  end if;
end process;

```

```
    end if;

    end if;
end process accessMem;

-----
-- Process handling of audio clock, sampling on sync
-- Output to ST Bus
-----
sample_buf_pro : process (csi_AudioClk12MHz_clk, csi_AudioClk12MHz_reset_n)

type state_type is (idle, validHigh);
variable valid_state : state_type;

begin

if csi_AudioClk12MHz_reset_n = '0' then -- asynchronous reset (active low)

    ast_source_data <= (others => '0');
    left_sample <= (others => '0');
    right_sample <= (others => '0');

elsif rising_edge(csi_AudioClk12MHz_clk) then -- rising clock edge

    -- Left channel
    if coe_AudioSync_export = '1' and AudioSync_last = '0' then
        left_sample <= coe_AudioIn_export;
        if (mute_left = '1') then
            ast_source_data <= (others => '0');
        else
            ast_source_data <= left_sample;
            valid_state := validHigh;
            valid_high <= '1';
            ast_source_channel <= chNrLeft;
        end if;
    end if;

    -- Right channel
    if coe_AudioSync_export = '0' and AudioSync_last = '1' then
        right_sample <= coe_AudioIn_export;
        if (mute_right = '1') then
            ast_source_data <= (others => '0');
        else
            ast_source_data <= right_sample;
            valid_state := validHigh;
            ast_source_channel <= chNrRight;
        end if;
    end if;

    case valid_state is
        when idle =>
            ast_source_valid <= '0';
        when validHigh =>
            ast_source_valid <= '1';
            valid_state := idle;
    end case;

    AudioSync_last <= coe_AudioSync_export;
```

```
end if;

end process sample_buf_pro;

st_bus_data_in : process (csi_AudioClk12MHz_clk, csi_AudioClk12MHz_reset_n)
begin
if csi_AudioClk12MHz_reset_n = '0' then -- asynchronous reset (active low)

    coe_AudioOut_export <= (others => '0');

elsif falling_edge(csi_AudioClk12MHz_clk) then -- falling clock edge
    if ast_sink_valid = '1' then
        coe_AudioOut_export <= ast_sink_data;
    end if;
end if;

end process st_bus_data_in;

end behaviour;
```

Kode 1 VHDL audio sync interface til ST bus (`audio_process_st2.vhd`)

5 Komponent beskrivelser og verifikation

Komponenter beskrevet i de efterfølgende afsnit tager udgangspunkt i signalbehandlingsteorien med en kort introduktion til MATLAB modellen og C-kode. Fokus for dette projekt er implementeringen i VHDL samt simulering og funktionel verifikation med ModelSim. De forskellige versioner af komponenten med optimerings tiltag, er beskrevet for optimering af area eller speed. Der er taget udgangspunkt i fagets teori omfattende emner som brug af "Pipelining" eller "Rolling up the pipeline". Hvert afsnit er afsluttet med en opsummering af komponentens FPGA ressource forbrug, latency eller throughput.

5.1 ModelSim Testbench

Dette kapitel beskriver kort den sekventielle testbench, der er skrevet for at test audio komponenterne omfattende ST bus interfacet med enten LMS filter, audio delay eller sigma delta converter. Der indlæses tekst filer med input samples for henholdsvis højre og venstre audio kanal, specificeret med **generic**. Det er vist i nedenstående testbench for LMSFilteret. Processen **WaveGen_Proc** (se VHDL koden på de næste sider) simulerer interfacet til **Codeclnterface** og læser samples fra filerne, som sendes til instanser af komponenterne: **audio_process_st** og **audiolmsfilterOpt_st**. Simuleringen stopper automatisk når alle data fra filerne er læst, med signalet **stop_the_clock**. Resultatet gemmes i tekst filer (**leftoutlms.txt**) med samples, hvor indholdet kan sammenlignes med en "golden" reference model fra MATLAB.

```
entity audiolmsfilterOpt_st_tb is

generic (
    filterOrder : natural := 64; -- Order of LMS filter
    audioWidth : natural := 24; -- 24 bit audio data
    -- Left audio channel number
    chNrLeft: std_logic_vector(2 downto 0) := "000";
    -- Right audio channel number
    chNrRight: std_logic_vector(2 downto 0) := "001";
```

```
-- Contains noise (x = LMS input)
leftin_name: string := "NoiseHex.txt";
-- Contains noise + sound (d = LMS desigeret)
rightin_name: string := "NoiseSignalHex.txt";
leftout_name: string := "leftoutlms.txt";
rightout_name: string := "rightoutlms.txt"
);

end audiolmsfilterOpt_st_tb;

architecture behaviour of audiolmsfilterOpt_st_tb is

!!!!!! Code removed - more details see audiolmsfilterOpt_st_tb.vhd

-- component instantiation for sync audio to ST bus converter
UUT: audio_process_st2
generic map ( audioWidth => audioWidth,
              chNrLeft => chNrLeft,
              chNrRight => chNrRight )
port map ( csi_AudioClk12MHz_clk      => Clk12Mhz,
            csi_AudioClk12MHz_reset_n => Reset,
            coe_AudioIn_export       => Audioin,
            coe_AudioOut_export     => AudioOut,
            coe_AudioSync_export    => Clk48KHz,
            csi_clockreset_clk       => Clk,
            csi_clockreset_reset_n  => Reset,
            avs_s1_write             => avs_write,
            avs_s1_read              => avs_read,
            avs_s1_chipselect        => avs_cs,
            avs_s1_address           => avs_address,
            avs_s1_writedata         => avs_writedata,
            avs_s1_readdata          => avs_readdata,
            ast_source_valid         => ast_input_valid,
            ast_source_data          => ast_input_data,
            ast_source_channel       => ast_input_channel,
            ast_sink_valid           => ast_output_valid,
            ast_sink_data             => ast_output_data,
            ast_sink_channel          => ast_output_channel);

-- component instantiation and optimized LMS filter
DUT: audiolmsfilterOpt_st
generic map (
  filterOrder => filterOrder,
  coefWidth => audioWidth, -- Keep coefficients same size as audio data
  audioWidth => audioWidth,
  chNrLeft => chNrLeft,
  chNrRight => chNrRight
)
port map (
  csi_AudioClk12MHz_clk      => Clk12Mhz,
  csi_AudioClk12MHz_reset_n  => Reset,
  ast_source_data             => ast_output_data,
  ast_source_valid            => ast_output_valid,
  ast_source_channel          => ast_output_channel,
  ast_sink_data               => ast_input_data,
  ast_sink_valid              => ast_input_valid,
  ast_sink_channel             => ast_input_channel,
  csi_clockreset_clk          => Clk,
```

```
    csi_clockreset_reset_n      => Reset,
    avs_s1_write                => avs_write,
    avs_s1_read                 => avs_read,
    avs_s1_chipselect           => avs_cs,
    avs_s1_address              => avs_address,
    avs_s1_writedata            => avs_writedata,
    avs_s1_readdata             => avs_readdata
);

-- Processes generating clocks
clocking: process --12Mhz
begin
  while not stop_the_clock loop
    Clk12Mhz <= '0', '1' after period12M / 2;
    wait for period12M;
  end loop;
  wait;
end process;

clocking_sync: process --48KHz
begin
  while not stop_the_clock loop
    Clk48KHz <= '0', '1' after period48K / 2;
    wait for period48K;
  end loop;
  wait;
end process;

clocking_50MHz: process
begin
  while not stop_the_clock loop
    Clk <= '0', '1' after period50M / 2;
    wait for period50M;
  end loop;
  wait;
end process;

Reset <= '0', '1' after 125 ns;
-- waveform generation
WaveGen_Proc: process
  -- files
  variable line: LINE;
  variable data: integer;
  variable val: signed(31 downto 0);
  variable i: integer;
  file leftinfile: TEXT open read_mode is leftin_name;
  file rightinfile: TEXT open read_mode is rightin_name;
  file leftoutfile: TEXT open write_mode is leftout_name;
  file rightoutfile: TEXT open write_mode is rightout_name;
begin

  -- Open simulation files
  file_open(leftinfile, leftin_name);
  file_open(rightinfile, rightin_name);
  file_open(leftoutfile, leftout_name);
  file_open(rightoutfile, rightout_name);

  -- signal assignments
  wait until Reset = '1';
```

```
wait until Clk48KHz = '1';
wait until Clk12Mhz = '1';
wait until Clk = '1';

-- Samples in left channel defines loops
while not endfile(leftinfile) loop

    wait until Clk48KHz = '1'; -- Left channel
    readline(leftinfile, line); -- read next text line from file
    read(line, data, 16); -- convert hex (16) numbers to integer value
    -- convert to audio 24 bit
    Audioin <= std_logic_vector(TO_SIGNED(data, audioWidth));
    data := TO_INTEGER(signed(AudioOut));
    write(line, data, right, 0, decimal, false);
    writeline(leftoutfile, line);

    wait until Clk48KHz = '0'; -- Right channel
    readline(rightinfile, line); -- read next text line from file
    read(line, data, 16); -- convert hex (16) numbers to integer value
    -- convert to audio 24 bit
    Audioin <= std_logic_vector(TO_SIGNED(data, audioWidth));
    data := TO_INTEGER(signed(AudioOut));
    write(line, data, right, 0, decimal, false);
    writeline(rightoutfile, line);

end loop;

-- Read last samples
wait for period48K;
wait for period48K;

file_close(leftinfile);
file_close(rightinfile);
file_close(leftoutfile);
file_close(rightoutfile);

stop_the_clock <= true;

end process WaveGen_Proc;

end behaviour;
```

Kode 2 VHDL test bench for LMS filter med ST interface (audiolmsfilterOpt_st_tb.vhd)

5.2 FIR filter

Dette kapitel giver en kort introduktion til de forskellige FIR filter typer, vi har implementeret og testet med interface til audio sync interfacet. Vi har implementeret 3 forskellige typer, som beskrevet i [4] kapitel 3. De 3 versioner omfatter en direct form 1, som senere skal benyttes til LMS filteret. Et optimeret FIR filter, hvor vi benytter et FIR filters symmetri. Et transposed filter, for at prøve alle mulighederne. Disse filtre er testet med en ModelSim test bench, der simulere grænsefladen til CodecInterfacet. I det følgende vises VHDL implementeringen for en symetriske og transposed implementering af FIR filteret.

```
sample_buf_pro : process (csi_AudioClk12MHz_clk, csi_AudioClk12MHz_reset_n)
variable left_sample : std_logic_vector(audioWidth-1 downto 0);
variable right_sample : std_logic_vector(audioWidth-1 downto 0);
variable filtered_data_temp : prod_type;
```

```
variable temp : tap_type;
variable result : prod_type;
begin

  if csi_AudioClk12MHz_reset_n = '0' then -- asynchronous reset (active low)
    for tap_no in filterOrder downto 0 loop
      tap(tap_no) <= (others => '0');
    end loop;
    coe_AudioOut_export <= (others => '0');
    AudioSync_last <= '0';

  elsif falling_edge(csi_AudioClk12MHz_clk) then -- faling clock edge

    -- Left channel
    if coe_AudioSync_export = '1' and AudioSync_last = '0' then
      left_sample := coe_AudioIn_export;

      for tap_no in filterOrder downto 1 loop - Stage 1
        tap(tap_no) <= tap(tap_no - 1);
      end loop;
      tap(0) <= shift_right(signed(left_sample), 1); -- Use only 23 bits

      for tap_no in (filterOrder/2)-1 downto 0 loop - Stage 2
        temp := tap(tap_no) + tap(filterOrder - tap_no);
        prod(tap_no) <= to_signed(coeff(tap_no), coefWidth) * temp;
      end loop;
      prod(filterOrder/2) <= to_signed(coeff(filterOrder/2),
                                         coefWidth) * tap(filterOrder/2);

      result := (others => '0');
      for tap_no in (filterOrder/2) downto 0 loop
        result := result + prod(tap_no);
      end loop;

      filtered_data_temp := shift_right(result, 8);

      if (mute_left = '1') then
        coe_AudioOut_export <= (others => '0');
      else - Stage 3
        coe_AudioOut_export <= std_logic_vector(
          filtered_data_temp(audioWidth-1 downto 0));
      end if;
    end if;

    -- Right channel - left out
    AudioSync_last <= coe_AudioSync_export;

  end if;

end process sample_buf_pro;
```

Kode 3 VHDL symmetrisk FIR filter implementation (audiofilter.vdh)

```
entity audiotransposedfilter is
generic (filterOrder : natural := 10;
         coefWidth : natural := 8;
         audioWidth : natural := 24); -- Default value
```

```
----- Code left out

subtype coeff_type is integer range -128 to 127;
type coeff_array_type is array (0 to filterOrder) of coeff_type;

subtype prod_type is signed(audioWidth+coefWidth-1 downto 0);
type prod_array_type is array (0 to filterOrder) of prod_type;

subtype sum_type is signed(audioWidth+coefWidth-1 downto 0);
type sum_array_type is array (0 to filterOrder) of sum_type;

constant coeff : coeff_array_type := (4, 8, 18, 32, 43, 47, 43, 32, 18, 8, 4);
signal prod    : prod_array_type;
signal sum     : sum_array_type;

begin

sample_buf_pro : process (csi_AudioClk12MHz_clk, csi_AudioClk12MHz_reset_n)
  variable left_sample : std_logic_vector(audioWidth-1 downto 0);
  variable x_n : signed(audioWidth-1 downto 0);
  variable right_sample : std_logic_vector(audioWidth-1 downto 0);
  variable filtered_data_temp : sum_type;
  variable result : sum_type;
begin

  if csi_AudioClk12MHz_reset_n = '0' then -- asynchronous reset (active low)
    for tap_no in filterOrder downto 0 loop
      sum(tap_no) <= (others => '0');
      prod(tap_no) <= (others => '0');
    end loop;
    coe_AudioOut_export <= (others => '0');
    AudioSync_last <= '0';
  elsif falling_edge(csi_AudioClk12MHz_clk) then -- falling clock edge

    -- Left channel
    if coe_AudioSync_export = '1' and AudioSync_last = '0' then

      left_sample := coe_AudioIn_export;
      x_n := shift_right(signed(left_sample), 1); -- Use only 23 bits

      -- Pipelined transposed FIR implementation
      -- See slides "FPGA Signal Processing" page 7
      for tap_no in filterOrder downto 0 loop
        prod(tap_no) <= x_n * to_signed(coeff(tap_no), coefWidth); -- Stage 1
      end loop;

      sum(filterOrder) <= prod(filterOrder);
      for tap_no in filterOrder-1 downto 1 loop
        sum(tap_no) <= sum(tap_no+1) + prod(tap_no+1); -- Stage 2
      end loop;

      result := prod(0) + sum(1); -- Stage 3
      filtered_data_temp := shift_right(result, 8);

      if (mute_left = '1') then
        coe_AudioOut_export <= (others => '0');
      else -- Stage 2

```

Komponenter til lydbehandling i en FPGA

```
coe_AudioOut_export <= std_logic_vector(
                                filtered_data_temp(audioWidth-1 downto 0));
end if;
end if;

-- Right channel left out
AudioSync_last <= coe_AudioSync_export;

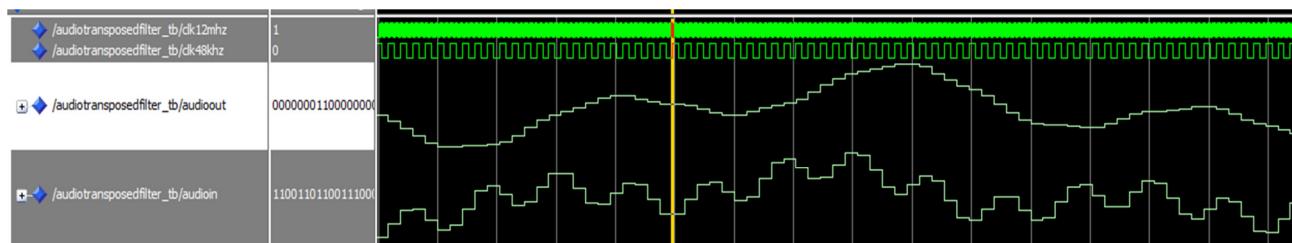
end if;

end process sample_buf_pro;
```

Kode 4 VHDL transposed FIR filter implementation (audiotransposed.vhd)

Begge implementationer benytter 12 MHz som aflæser, hvornår audio sync signalet indikeret et nyt sample på henholdsvis højre og venstre audio kanal er klar. Begge versioner udfører beregningen med en pipeline på 3. Det betyder, at der bliver en latency på unødvendigt 3 audio sync perioder (48 KHz). Filteret kunne dog nemt afvikles med en audio clock på op til de 12 MHz.

Det symmetriske filter bruger ca. halvt så mange multiplikationer og additioner som direkte form 1. Det transposed filter slipper for shift af delay line og benytter kun én multiplikation og addition for hver tap i filteret. Begge filter kan konfigureres med **generic** til fastsættelse af filterorden, bredde af audio og koefficienterne. Nedenfor er vist resultatet for simulering i ModelSim med tilhørende test bench. For yderligere detaljer af kode og test benches henvises til appendix B.



Figur 6 Transposed lavpass FIR filtered data, audio out øverst

5.3 LMS filter

Et adaptivt filter er et digitalt FIR filter, hvor filterets koefficienter automatisk justeres af en algoritme i dette tilfælde LMS "Least Mean Squares". For flere detaljer om adaptiv filter teori se kapitel 4.4 [1].

Princippet er illustreret nedenfor, hvor input signalet $x(n)$ filtreres med det digitale FIR filter. Det ønskede signal $d(n)$ subtraheres fra det filtrerede signal $y(n)$. Fejlen $e(n)$ benyttes til at opdaterer koefficienterne i FIR filteret.

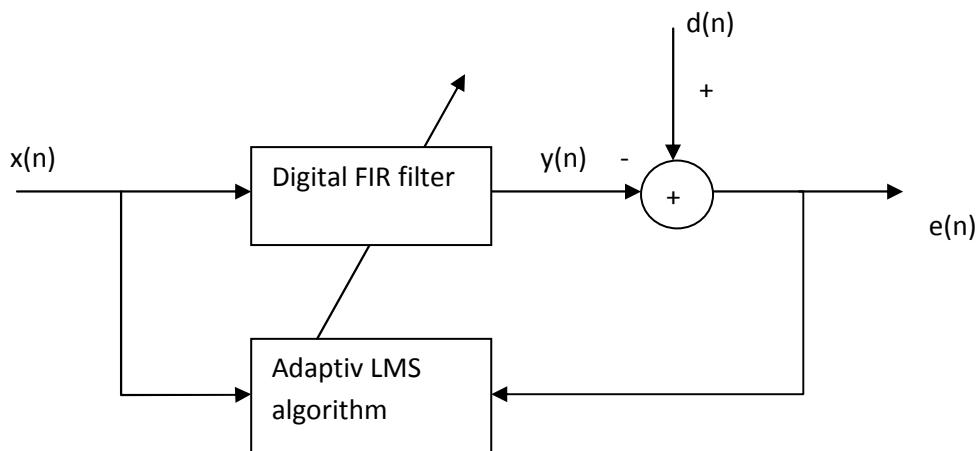


Figure 1 Adaptiv LMS filter

Det digitale FIR filter beregnes som

$$y(n) = \sum_{l=0}^{L-1} w(n)x(n-l)$$

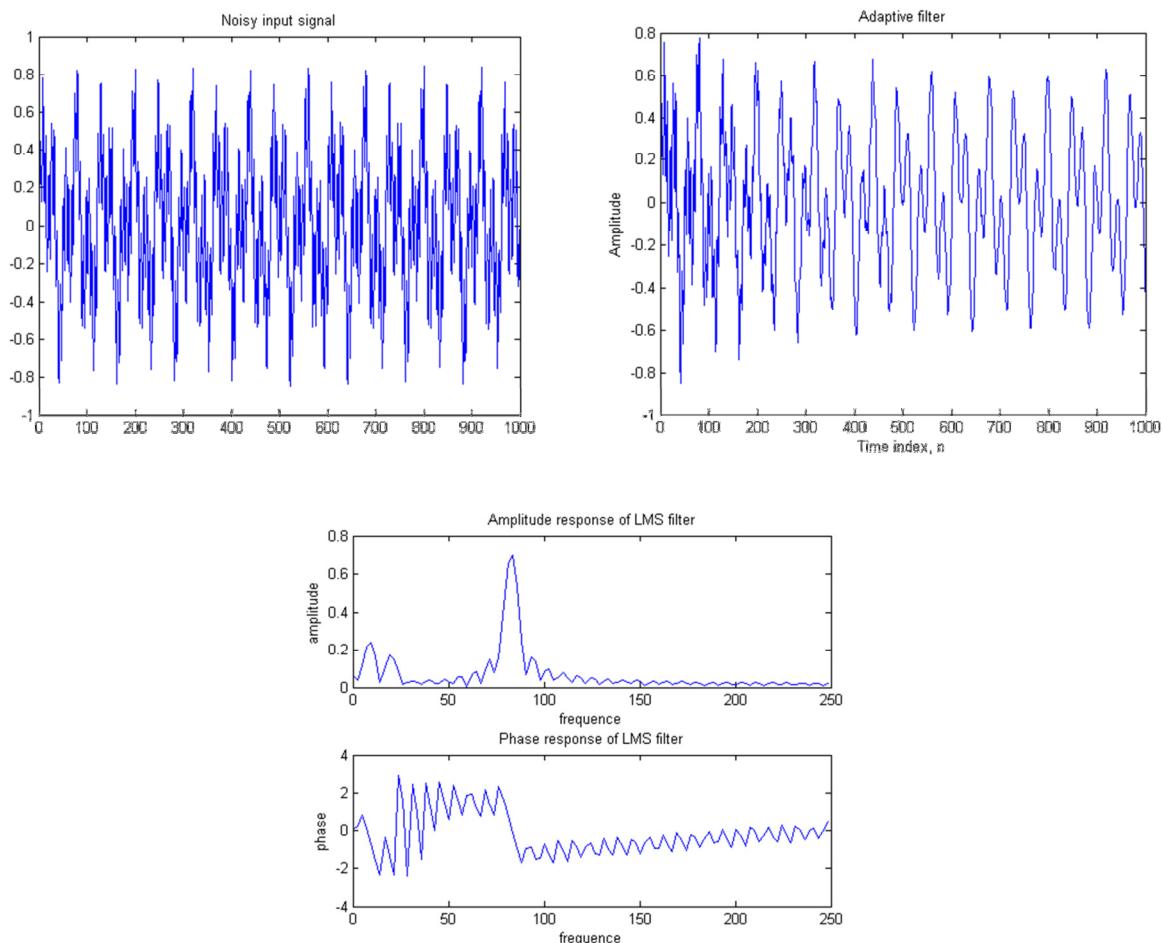
LMS algoritmen beregner nye koefficienter efter formlen, hvor μ er en meget lille adaptions konstant

$$\mathbf{W}(n+1) = \mathbf{W}(n) - \mu \mathbf{X}(n)e(n)$$

Fejl signalet beregnes som forskellen mellem det ønskede signal $d(n)$ og det filtrerede $y(n)$

$$e(n) = d(n) - y(n)$$

Ønsket er at implementerer et LMS filteret med DE2 boardet, hvor venstre audio kanalen indeholder et ønsket støj signal $d(n)$ og højre kanal indeholder et ønsket signal med støj $x(n)$. Output $e(n)$ vil så indeholde det ønskede signal uden støj se nedenstående figur. (Se MATLAB model for flere detaljer: [LMSNoiseSuppressionSolution.m](#)) Nedenstående simulering viser resultatet med en filteorden på 64 med adaptions konstant på 0.004. Signalet indeholder støj blandet med sinus signaler 0.8 KHz, 1 KHz og 8 KHz med en samplingsfrekvens på 48 KHz.



Figur 7 Fra højre øverst vises ønsket signal med støj og et 8 kHz uønsket signal $x(n)$, til venstre vises det filtrerede signal $e(n)$, nederst vises amplitude og frekvens responsen for LMS filter koefficienterne

Implementeringen i VHDL tager udgangspunkt i MATLAB og C-kode se nedenfor.

```
// Shift delay line
for(k=len-1; k > 0; k--)
    dly[k] = dly[k-1];

// Insert next x
dly[0] = x;

// Convolution: w * x
for(k=0; k < len; k++)
    yn += wgt[k] * dly[k];

// Calculate output result
out = (yn >> 15);

// Estimate error (n)
err = d - out;

// Adjust weights
for(k=0; k < len; k++)
{
    wk_i = err*dly[k];
```

```

    wk_s = (wk_i >> 15); // Truncate
    wk_i = adpt*wk_s;
    wgt[k] += (wk_i >> 15); // Truncate
}

```

Kode 5 C-koden for 16 bits LMS filter (audiolmsfilterOpt st tb.vdh)

C-koden viser LMS filterets for en 16-bits fixed point implementering. Først skiftes delay line, herefter udføres FIR filtreringen. Den estimerede fejl beregnes og til slut beregnes nye koefficienter ($w(n)$) efter LMS algoritmen.

5.3.1 FPGA Implementering

Dette afsnit omfatter hvordan LMS filteret er implementeret for et FPGA design. Der er beskrevet 2 versioner optimeret for henholdsvis speed og area som beskrevet i kapitel 1 og 2 i [3]. I forhold til speed er målet at have en "low latency" med et pipeline design, hvor der er foretaget "unrolling" af for løkkerne. Den næste version er optimeret for area, med en audio samplingsfrekvens på 48 KHz har vi rigeligt med cykler derfor er der foretaget "rolling up the pipeline".

Den første version vi har implementeret i VHDL benytte 24 bit i stedet for 16 bit. Denne første version (Appendix - audiolmsfilter.vhd) er testet med I2S til Sync Bus interfacet. Versionen beregner LMS filteret i én 12 mHz clock cycle og er pipelined med 3 stages. Denne version er først udviklet og testet med ModelSim og en tilhørende test bench er udviklet. Den indlæser test signaler fra filer generet fra MATLAB. Målet med denne version er sikre en korrekt implementering i VHDL. Alle midlertidige multiplikations resultater er gemt med en oplosning på 48 bits. Denne version er testet med en filterorden på 10. Versionen er optimeret for speed, men absolut ikke area. I nedenstående VHDL proces (**sample_buf_pro**) er koden vist for implementationen af filteret. Processen aflæser det 48 kHz sync signal (**AudioSync**) som med skift fra lav til høj indikerer et nyt sample på venstre audio kanal (**noise_sample**), ved skift fra høj til lav aflæses højre audio kanal (**sound_sample**). Denne sekventielle proces benytter en 12 MHz clock og den 48 kHz audio sync virker som et enable signal.

```

-- Process using audio clock (12 mHz) to sample sync signal (48 kHz)
-- Performs LMS filtering in one 12 mHz clock cycle
-----
sample_buf_pro : process (csi_AudioClk12MHz_clk, csi_AudioClk12MHz_reset_n)
  variable noise_sample : std_logic_vector(audioWidth-1 downto 0);
  variable sound_sample : std_logic_vector(audioWidth-1 downto 0);
  variable result : prod_type;
  variable filtered_result : prod_type;
  variable wk_i : signed((2*audioWidth)-1 downto 0);
  variable error : tap_type;
  variable wk_ii : signed(audioWidth+coefWidth-1 downto 0);
begin
  if csi_AudioClk12MHz_reset_n = '0' then -- asynchronous reset (active low)
    for tap_no in filterOrder downto 0 loop
      coeff(tap_no) <= (others => '0');
      tap(tap_no) <= (others => '0');
      prod(tap_no) <= (others => '0');
      wk_s(tap_no) <= (others => '0');
    end loop;
    noise_sample := (others => '0');
  end if;
  sound_sample <= noise_sample;
  result <= prod;
  filtered_result <= result;
  error <= abs(sound_sample - filtered_result);
  wk_i <= signed(error);
  wk_ii <= signed(result);
  for i in filterOrder-1 downto 0 loop
    if i = 0 then
      coeff(i) <= wk_ii;
      tap(i) <= wk_i;
      prod(i) <= result;
      wk_s(i) <= noise_sample;
    else
      coeff(i) <= (others => '0');
      tap(i) <= (others => '0');
      prod(i) <= (others => '0');
      wk_s(i) <= (others => '0');
    end if;
  end loop;
  noise_sample <= (others => '0');
end process;

```

```
sound_sample := (others => '0');
error := (others => '0');
AudioSync_last <= '0';
coe_AudioOut_export <= (others => '0');

elsif falling_edge(csi_AudioClk12MHz_clk) then -- falling clock edge

    -- Left channel
    if coe_AudioSync_export = '1' and AudioSync_last = '0' then

        noise_sample := coe_AudioIn_export; -- Noise signal

        -- Direct FIR filter pipelined - 3 stages for LMS filter
        -- First stage shift delayline - stage 1
        for tap_no in filterOrder downto 1 loop
            tap(tap_no) <= tap(tap_no - 1);
        end loop;
        tap(0) <= signed(noise_sample);

        -- Performs MAC for FIR filter
        result := (others => '0');
        for tap_no in filterOrder downto 0 loop
            result := (coeff(tap_no) * tap(tap_no)) + result;
        end loop;
        filtered_result := shift_right(result, audioWidth-1);
        error := signed(sound_sample) - resize(filtered_result, audioWidth);

        -- Performs adjust LMS algorithm of weights - stage 2 and 3
        for tap_no in filterOrder downto 0 loop
            wk_i := error * tap(tap_no);
            wk_s(tap_no) <= resize(shift_right(wk_i, audioWidth-1), audioWidth);
            wk_ii := adptStep * wk_s(tap_no);
            coeff(tap_no) <= coeff(tap_no) + resize(shift_right(wk_ii,
                                                                    audioWidth-1), coefWidth);
        end loop;

        -- Output LMS filtered left channel
        if (mute_left = '1') then
            coe_AudioOut_export <= (others => '0');
        else
            coe_AudioOut_export <= std_logic_vector(error(audioWidth-1 downto 0));
        end if;
    end if;

    -- Right channel
    if coe_AudioSync_export = '0' and AudioSync_last = '1' then
        sound_sample := coe_AudioIn_export;
        if (mute_right = '1') then
            coe_AudioOut_export <= (others => '0');
        else
            coe_AudioOut_export <= sound_sample;
        end if;
    end if;

    AudioSync_last <= coe_AudioSync_export;

end if;

end process sample_buf_pro;
```

Kode 6 VHDL LMS filter med audio sync interface (`audiolmsfilter.vhd`)

Først skiftes "Tapped Delay Line" for FIR filteret, der er implementeret i "Direct Form". Det er ikke muligt at benytte hverken en "Transposed Structure" eller en symmetrisk implementering. Den "Tapped Delay Line" skal efterfølgende bruges i udregningen af de nye koefficienter. Det pipelined filter består af 3 stages. Første stage er skift af delay line. Anden og tredje stage er udregning af FIR filteret samt opdatering af koefficienterne. Filteret udføres med en frekvens på 48 kHz med en latency på 2 samples (41.2 us). Denne første version kræver mange FPGA ressourcer og kunne fint køre med en samplingsfrekvens helt op til de 12 MHz.

Den næste version af LMS filteret, vi har implementeret, benytter i stedet ST bus interfacet og er nu optimeret for area. En state maskine er implementeret, som med en 12 MHz clock fortager "Rolling up the pipeline". Denne state maskine er implementeret i en separat process, bestående af en række states, der initieres når nye samples modtages. Denne proces bruger væsentlige færre FPGA ressourcer især multipliers og adders. Denne version benytter kun 4x24 bits multipliers og 1x48 bits adder og 2x24 bits adders. Antallet af multipliers og adders er uafhængig af filterets længde. Hver state udføres synkront med 12 MHz clock signalet. Med en filter længde på 64, tager filteret $2 \times 64 + 3$ clocks for udregningen. Med 12 MHz tager det ca. 11 us.

```
-->>>-----<--><----->>>
-- This process performs LMS filtering, optimized for area
-- Performs LMS filtering in filterOrder*2 + 3 clocks (12 MHz)

LMSFilter : process (csi_AudioClk12MHz_clk, csi_AudioClk12MHz_reset_n)
  variable result : prod_type;
  variable filtered_result : prod_type;
  variable wk_i : signed((2*audioWidth)-1 downto 0);
  variable wk_ii : signed(audioWidth+coefWidth-1 downto 0);
  variable wk_s : tap_array_type;
  variable tap_no : index_type;
  variable error : tap_type; -- Output result from LMS filter
begin

  if csi_AudioClk12MHz_reset_n = '0' then

    for tap_no in filterOrder downto 0 loop
      coeff(tap_no) <= (others => '0');
      tap(tap_no) <= (others => '0');
      prod(tap_no) <= (others => '0');
      wk_s(tap_no) := (others => '0');
    end loop;
    error := (others => '0');
    output_sample <= (others => '0');
    filter_state <= idle;

  elsif rising_edge(csi_AudioClk12MHz_clk) then -- faling clock edge

    case filter_state is

      when idle =>
        if process_sample = '1' then
```

```
    input_sample <= signed(noise_sample);
    filter_state <= step1;
end if;

when step1 =>
  -- Direct FIR filter
  -- Shift delayline
  for no in filterOrder downto 1 loop
    tap(no) <= tap(no - 1);
  end loop;
  tap(0) <= input_sample;
  tap_no := filterOrder;
  result := (others => '0');
  filter_state <= step2;

when step2 =>
  -- Direct FIR filter
  -- Performs MAC for FIR filter
  result := (coeff(tap_no) * tap(tap_no)) + result;
  if (tap_no = 0) then
    filter_state <= step3;
  else
    tap_no := tap_no - 1;
  end if;

when step3 =>
  -- Computes error
  filtered_result := shift_right(result, audioWidth-1);
  error := signed(sound_sample) - resize(filtered_result, audioWidth);
  tap_no := filterOrder;
  filter_state <= step4;

when step4 =>
  -- Performs adjust LMS algorithm of weights, 2 stages pipelining
  wk_i := error * tap(tap_no);
  wk_s(tap_no) := resize(shift_right(wk_i, audioWidth-1), audioWidth);
  wk_ii := adptStep * wk_s(tap_no);
  coeff(tap_no) <= coeff(tap_no) + resize(shift_right(wk_ii,
    audioWidth-1), coefWidth);
  if (tap_no = 0) then
    filter_state <= idle;
  else
    tap_no := tap_no - 1;
  end if;

when others =>
  filter_state <= idle;

end case;

output_sample <= error;

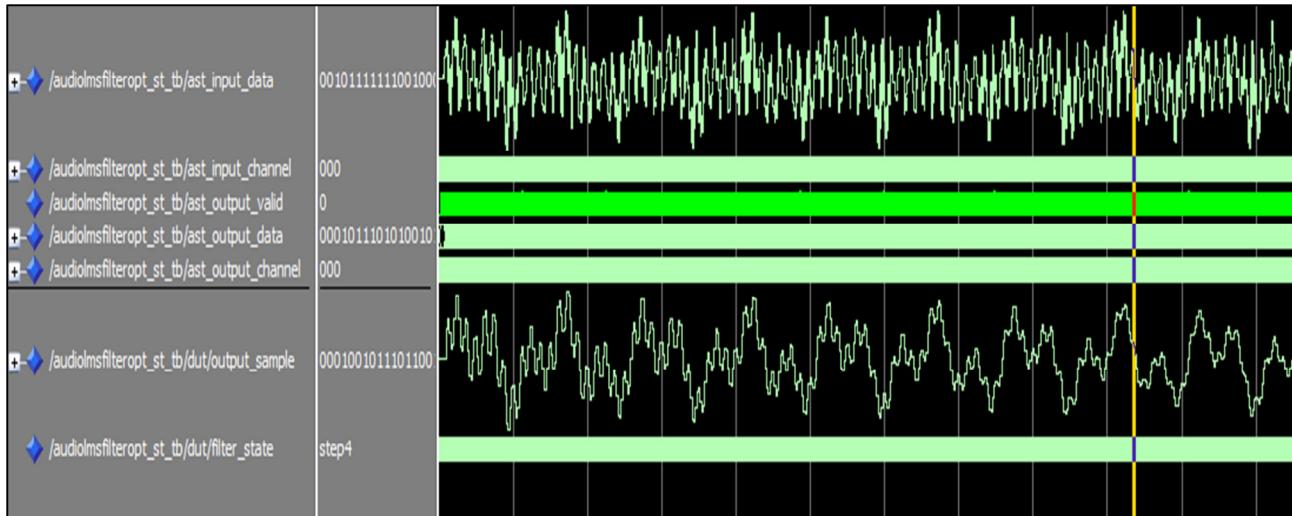
end if;

end process LMSFilter;
```

Kode 7 VHDL LMS filter med ST interface optimeret for area (audiolmsfilterOpt_st.vhd)

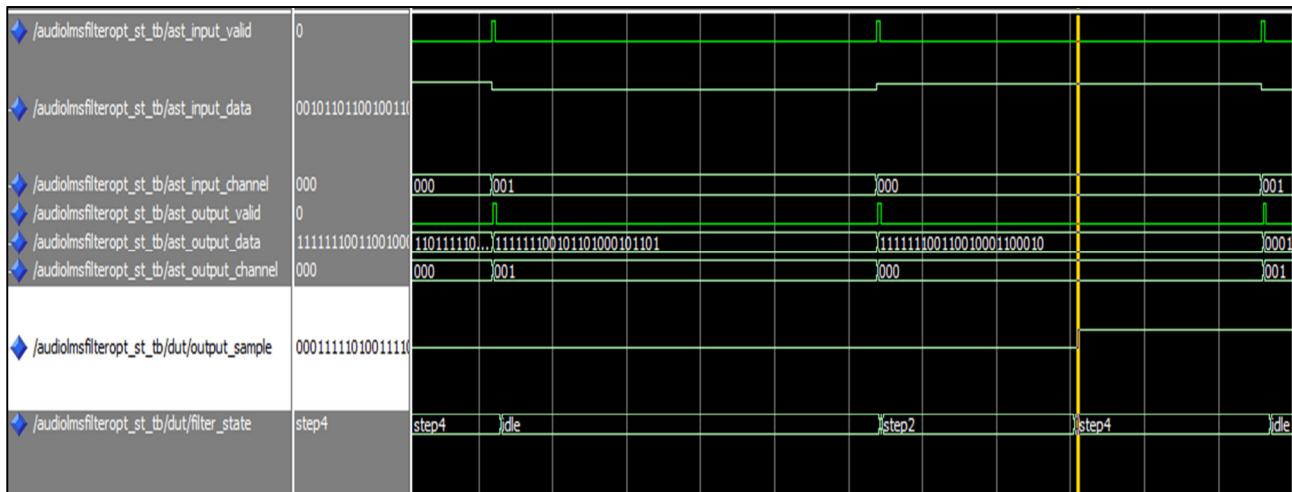
Komponenter til lydbehandling i en FPGA

Filteret er testet med den sekventiel testbench i ModelSim beskrevet i 5.1. Simuleringen udføres for ca. 21 ms afspilning af 1000 samples, som er generet af MATLAB. Figur 9 viser resultatet af timingsimuleringer i ModelSim, hvor det kan ses på **output_sample** signalet at LMS filteret justerer sig på plads.



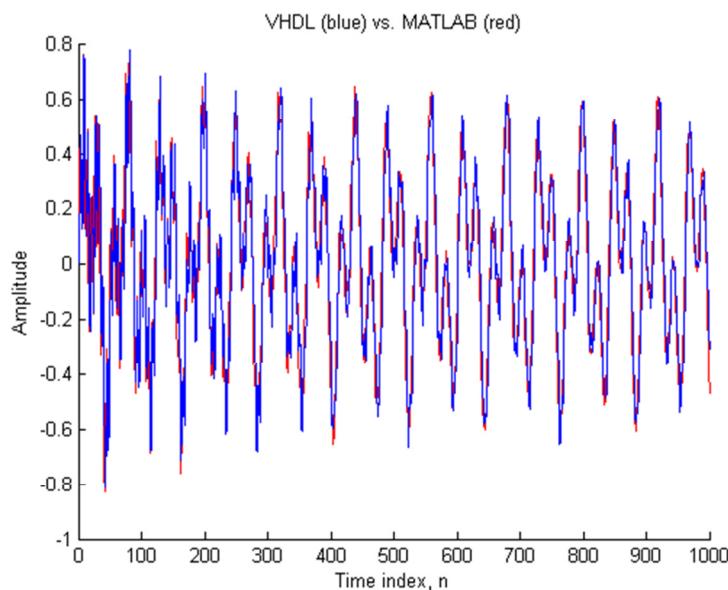
Figur 8 Verifikation af optimeret LMS filter med ST bus interface

Figur 10 viser timing for, hvordan valid signalene på ST bussen indikerer, at et nyt sample er klart for henholdsvis højre og venstre kanal. For **output_channel** = 0 (venstre), starter LMS state maskinen med at gennemløbe sine filter states, her er et nyt **output_sample** klart efter **step3** markeret med den gule cursor. I **step 4** adapteres koefficienterne, hvorefter LMS filteret går tilbage til **idle** state og er igen klar til at beregne det næste sample.



Figur 9 LMS filter med ST bus interface – audio samples for højre og venstre kanal

Figur 10 viser outputtet fra det simulerede LMS filter med 24 bits fixed point sammenlignet med versionen i double præcision fra MATLAB:



Figur 10 Signal efter LMS filtrering i ModelSim (blå) sammenlignet med MATLAB model (rød)

Nedenstående tabel viser FPGA ressource forbruget for de 2 version af LMS filteret. Dette er målt ved at syntetisere LMS komponenten i et selvstændigt Quartus projekt.

| LMS Filter (24 bit width) | Filter order | Latency (12 MHz) | Optimized for | Multipliers (9 bit) | LE (DE2) | Registers | Restricted Fmax-12M |
|--|-----------------|---------------------|------------------|------------------------|-------------------|-----------|------------------------|
| LMS speed audiolmsfilter.vhd (Sync bus) | 10 | 250 ns | speed | 70 | 11354 (34) | 876 | 28.66 MHz |
| LMS area audiolmsfilterOpt _st.vhd (ST bus) | 64 | 16 us | area | 16 | 6832 (21) | 4898 | 28.05 MHz |

Tabellen ovenfor viser, at LMS filteret optimeret for area med en orden 64 tappe kan håndtere op til en samplingsfrekvens på 91.6 KHz. Den bruger samlet ca. 21 % af de logiske elementer af FPGA'en på DE2 boardet. Med en samplingsfrekvens på 48 KHz er den teoretiske maksimale filterorden ca. 123 tappe. Der er en fin balance mellem brug af LE og registeres, med ca. det samme antal, hvilket betyder at flipflops i FPGA'en også benyttes. Den første version af filteret optimeret for hastighed benytter mange FPGA ressourcer. Med en filterorden på kun 10 bruger den alle 70 multipliers og hele 34% af FPGA'ens logiske elementer. De multipliers, der ikke er plads til, er implementeret i LE blokke. Det er til gengæld et meget hurtigt filter, der har et throughput på 24 bit * 12 MHz = 288 Mbit/sec, med en latency på 250 ns. Teoretisk er det maksimale throughput på 24 bit * 28.66 MHz = 687 Mbit/sec.

5.4 Delay

Vi har valgt at implementere et simplet stereo delay. Målet er at finde ud af, hvordan de interne ram blokke kan bruges direkte fra VHDL koden. Side 11-20 [2], beskriver hvordan en Dual-Port synkron RAM block kan skrives i VHDL, vi har implementeret en modificeret version vist i nedenstående kode eksempel.

```
ENTITY delay_ram IS
    GENERIC (
        bitWidth : natural := 24;
        ramSize : natural := 2048
    );
    PORT (
        clock: IN STD_LOGIC;
        data: IN STD_LOGIC_VECTOR (bitWidth-1 DOWNTO 0);
        write_addr: IN INTEGER RANGE 0 TO ramSize-1;
        read_addr: IN INTEGER RANGE 0 TO ramSize-1;
        we: IN STD_LOGIC;
        q: OUT STD_LOGIC_VECTOR (bitWidth-1 DOWNTO 0)
    );
END delay_ram;

ARCHITECTURE rtl OF delay_ram IS
    TYPE MEM IS ARRAY(0 TO ramSize-1) OF STD_LOGIC_VECTOR(bitWidth-1
DOWNTO 0);
    SIGNAL ram_block: MEM;
BEGIN

    PROCESS (clock)
    BEGIN
        IF (clock'event AND clock = '1') THEN
            IF (we = '1') THEN
                ram_block(write_addr) <= data;
            END IF;
            q <= ram_block(read_addr);
        END IF;
    END PROCESS;
END rtl;
```

Kode 8 VHDL delay ram implementation med FPGA block ram (delay_ram.vhd)

Et nyt component til ST bussen er designet, der benytter en instans af ovenstående delay block for hver audio kanal. Funktionen kan bypasses fra softwaren via. MM bussen. Kun den essentielle kode er vist for instantiering af delay ram med en ST bus sink og source implementation for venstre kanal. Princippet er at indlæse nye samples til RAM blokken og udlæse den ældste værdi. En separat proces **sample_st_source** håndterer afsending af data fra RAM blokken fra ST source interfacet. Vi har valgt at afsende data på ST bussen med rising edge (Source) og sample på falling edge (Sink). Dermed sikres at en kæde af ST komponenter altid vil overfører data korrekt, hvor **ast_sink_valid** signalet bruges til synkronisering.

```
entity audiodelay_st is
    generic (delaySize : natural := 2048;
             audioWidth : natural := 24;
             chNrLeft : std_logic_vector(2 downto 0) := "000";
             chNrRight : std_logic_vector(2 downto 0) := "001");
    port (
        -- Clock Interface - left out
```

```
-- ST Bus - left out
-- MM Bus - left out
);

end audiodelay_st;

architecture behaviour of audiodelay_st is

-- Signals and constant declarations left out

begin

DelayRAMLeft: entity work.delay_ram
generic map ( bitWidth => audioWidth,
              ramSize => delaySize )
port map ( clock => csi_AudioClk12MHz_clk,
           data => lraminput,
           write_addr => lramwaddr,
           read_addr => lramraddr,
           we => lramwe,
           q => lramoutput);

DelayRAMRight: -- left out

-- Process handling of MM bus left out
-----
-- Process handling of audio clock, sampling of ST input sink data
-----
sample_st_sink : process (csi_AudioClk12MHz_clk, csi_AudioClk12MHz_reset_n)
begin

if csi_AudioClk12MHz_reset_n = '0' then -- asynchronous reset (active low)
    left_delay <= (others => '0');
    right_delay <= (others => '0');
    lraminput <= (others => '0');
    rraminput <= (others => '0');
    lramwaddr <= CI_START_WRITE_ADDR; -- start write
    rramwaddr <= CI_START_WRITE_ADDR;
    lramraddr <= CI_START_READ_ADDR;
    rramraddr <= CI_START_READ_ADDR;
    lramwe <= '0';
    rramwe <= '0';

elsif falling_edge(csi_AudioClk12MHz_clk) then -- falling clock edge

    rramwe <= '0';
    lramwe <= '0';

    -- New sample ready on ST bus
    if ast_sink_valid = '1' then

        -- Read audio channel
        case ast_sink_channel is

            when chNrLeft =>

                -- Left channel input
```

```
left_delay <= lramoutput;
lraminput <= ast_sink_data;

-- Write value to ram
lramwe <= '1';

if (lramwaddr < delaySize - 1) then
    -- Increment write address
    lramwaddr <= lramwaddr + 1;
else
    lramwaddr <= 0;
end if;

if (lramraddr < delaySize - 1) then
    -- Increment read address
    lramraddr <= lramraddr + 1;
else
    lramraddr <= 0;
end if;

when chNrRight =>

    -- Right channel input - left out

when others =>
    null;

end case;

end if;

end if;

end process sample_st_sink;

-- Handling of source channel
sample_st_source : process (csi_AudioClk12MHz_clk, csi_AudioClk12MHz_reset_n)
begin

    if csi_AudioClk12MHz_reset_n = '0' then -- asynchronous reset (active low)
        ast_source_data <= (others => '0');
        ast_source_channel <= (others => '0');
        ast_source_valid <= '0';

    elsif rising_edge(csi_AudioClk12MHz_clk) then -- rising clock edge

        ast_source_valid <= '0';

        -- New sample to left delay line
        if (lramwe = '1') then
            -- Left channel output
            if (bypass_left = CI_BYPASS) then
                ast_source_data <= lraminput;
            else
                ast_source_data <= left_delay; -- Output from delay line
            end if;

            ast_source_channel <= chNrLeft;
            ast_source_valid <= '1';

    end if
```

```

    end if;

    -- New sample to right delay line left out

    end if;

end process sample_st_source;

end behaviour;

```

Kode 9 VHDL audio delay med ST interface (audiodelay_st.vhd)

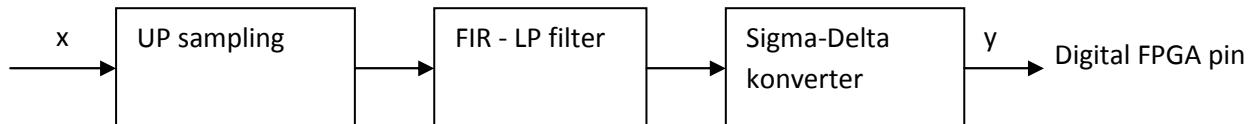
Delay filteret er testet i ModelSim med samme type testbench som beskrevet i kapitel 5.1 med ST interface. Nedenstående tabel viser FPGA ressource forbruget kompileret med Quartus, hvor der benyttes block ram. Bemærk hvor få LE blokke komponenten benytter. Det meste logik ligger i memory blokke.

| Delay Filter (24 bit width) | Delay size | Audio delay | Memory bits | Multipliers (9 bit) | LE (DE2) | Registers | Restricted Fmax-12M |
|---------------------------------------|---------------|----------------|-----------------|------------------------|--------------|-----------|------------------------|
| audiodelay_st.vhd (ST bus) | 2048 | 42 ms | 97152 (20 %) | 0 | 179 (1 %) | 124 | 156.79 MHz |

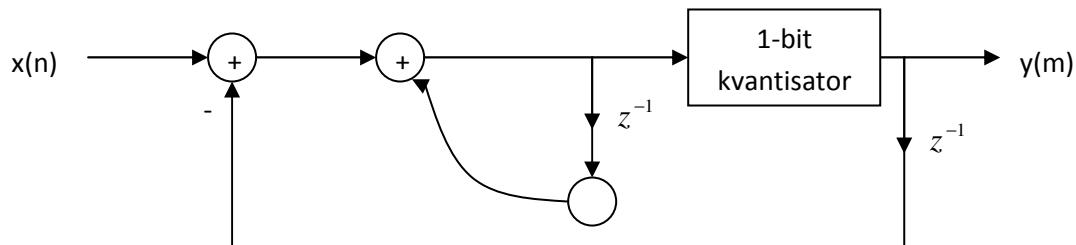
5.5 Sigma delta konverter

(Rune)

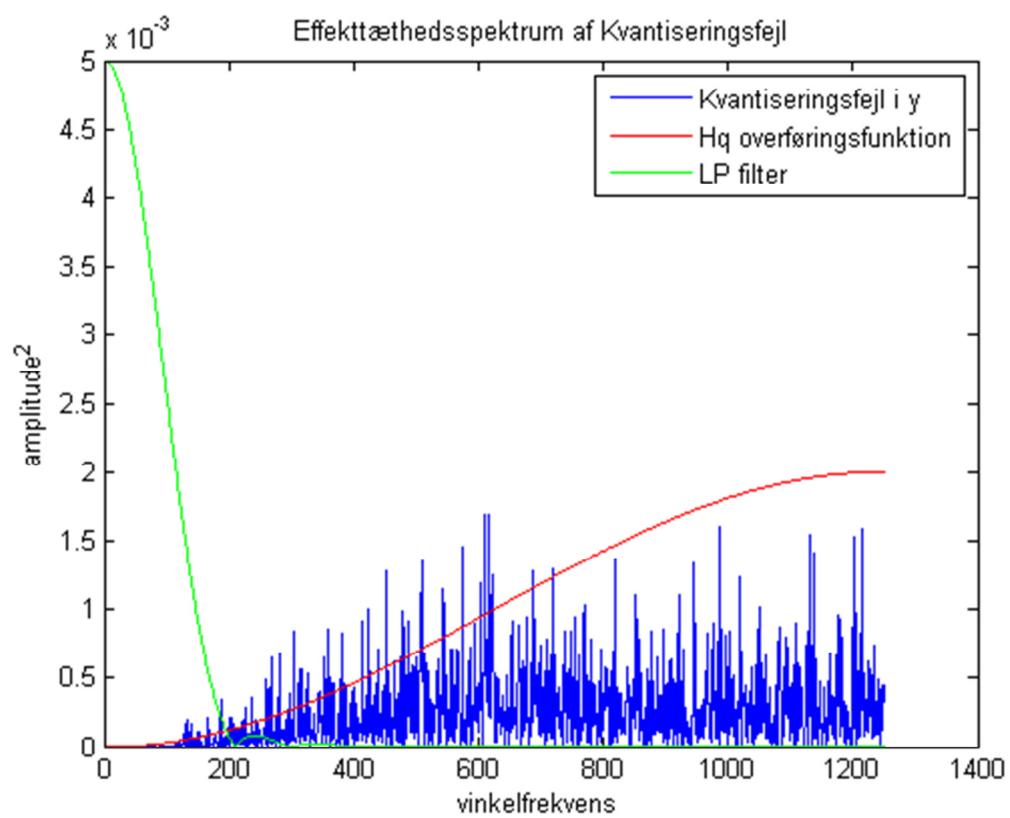
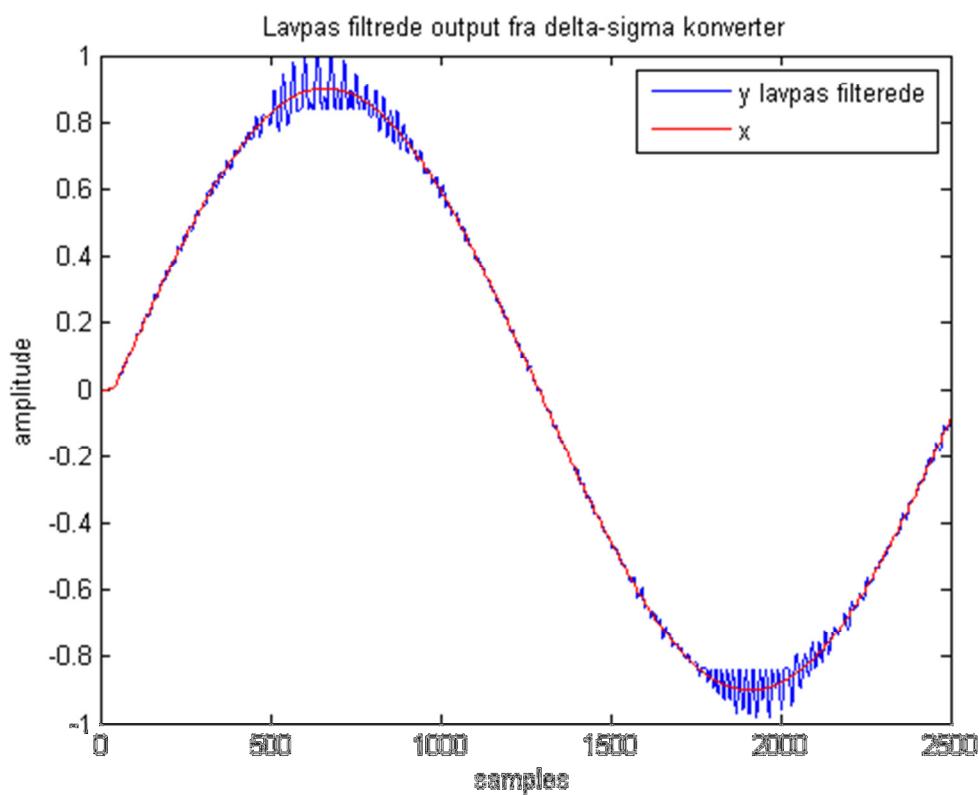
Dette afsnit er ikke beskrevet i detaljer kun illustreret med tegninger og VHDL kode – der henvises til MATLAB koden i appendix A samt VHDL koden i appendix B for yderligere detaljer.



Princippet er at oversample input signalet X, herefter foretages en lavpass filtrering. Selve sigma delta konverteren er illustreret i nedenstående figur med en 1-bits kvantisator og et noise shapings filter. Signalet X oversamples med 25 gange 48 KHz. En gated clock genereres ud fra 12 MHz audio clocken.



Komponenter til lydbehandling i en FPGA



```
library IEEE;
use IEEE.Std_logic_1164.all;
use IEEE.Numeric_Std.all;

entity UpSampler is
  generic (filterOrder : natural := 26;
           coefWidth   : natural := 8;
           inputWidth   : natural := 24);
  port (
    -- Common --
    clk          : in  std_logic;      -- 12MHz
    reset_n      : in  std_logic;

    -- ST Bus --
    ast_sink_data    : in  std_logic_vector(inputWidth-1 downto 0);
    ast_sink_valid   : in  std_logic;

    -- sigma dalta converter --
    outputPin       : out std_logic);

end entity UpSampler;

architecture code of UpSampler is

  signal last_data : signed(inputWidth-1 downto 0);
  signal last_data1 : signed(inputWidth-1 downto 0); --double flip flop

  -- Upsampler and FIR filter
  subtype coeff_type is integer range -128 to 127;
  type coeff_array_type is array (0 to filterOrder/2) of coeff_type;

  subtype tap_type is signed(inputWidth-1 downto 0);
  type tap_array_type is array (0 to filterOrder) of tap_type;

  subtype prod_type is signed(inputWidth+coefWidth-1 downto 0);
  type prod_array_type is array (0 to filterOrder/2) of prod_type;

  constant coeff : coeff_array_type :=
    (1, 1, 2, 3, 4, 6, 8, 11, 13, 15, 17, 18, 19, 19); -- Cut off 24 kHz

  signal tap    : tap_array_type ;
  signal prod   : prod_array_type ;

  -- Sigma dalta converter --
  signal sigmaIn : signed(inputWidth+1 downto 0);
  signal sigmaIntegratorLast1 : signed(inputWidth+1 downto 0);
  signal sigmaIntegratorLast2 : signed(inputWidth+1 downto 0);
  signal sigmaDac : signed(inputWidth+1 downto 0);
  signal outUpSampler : std_logic_vector(inputWidth-1 downto 0);

  -- Clk Divider --
  signal ClkDividerOut : std_logic; -- 12MHz / 10 = 1,2MHz

begin

  Filter : process(ClkDividerOut, reset_n)
```

```
-- FIR Filter
variable temp : tap_type;
variable result : signed(inputWidth+coefWidth-1 downto 0);
-- Sigma Delta Converter
variable sigmaSum : signed(inputWidth+1 downto 0);
variable sigmaIntegrator1 : signed(inputWidth+1 downto 0);
variable sigmaIntegrator2 : signed(inputWidth+1 downto 0);
variable sigmaOut : std_logic;
begin

if reset_n = '0' then

    for tap_no in filterOrder downto 0 loop
        tap(tap_no) <= (others => '0');
    end loop;
    for tap_no in filterOrder/2 downto 0 loop
        prod(tap_no) <= (others => '0');
    end loop;

    sigmaIn <= (others => '0');
    sigmaIntegratorLast1 <= (others => '0');
    sigmaIntegratorLast2 <= (others => '0');
    sigmaDac <= (others => '0');
    last_data <= (others => '0');

elsif rising_edge(ClkDivideOut) then

    for tap_no in filterOrder downto 1 loop
        tap(tap_no) <= tap(tap_no - 1);
    end loop;

    --double flip flop
    last_data <= last_data1;
    tap(0) <= last_data;

    for tap_no in (filterOrder/2)-1 downto 0 loop
        temp := resize(tap(tap_no), inputWidth) +
            resize(tap(filterOrder - tap_no), inputWidth);
        prod(tap_no) <= to_signed(coeff(tap_no),
            coefWidth) * temp;
    end loop;

    prod(filterOrder/2) <= to_signed(coeff(filterOrder/2),
        coefWidth) * tap(filterOrder/2);
    result := (others => '0');
    for tap_no in (filterOrder/2) downto 0 loop
        result := result + prod(tap_no);
    end loop;

    outUpSampler <= std_logic_vector(
        shift_right(result, 8)(inputWidth-1 downto 0));
    --^ ^ ^ ^ ^ ^ ^ ^ ^
    --| | | | | | | | |
    --Upsampler and FIRfilter
    -----
--Sigma delta converter
sigmaIn <= resize(
    shift_right(result, 8)(inputWidth-1 downto 0), inputWidth+2);
sigmaSum := sigmaIn - sigmaDac;
```

```
sigmaIntegrator1 := sigmaSum + sigmaIntegratorLast1;
sigmaIntegratorLast1 <= sigmaIntegrator1;

sigmaIntegrator2 := sigmaIntegrator1 - sigmaDac + sigmaIntegratorLast2;
sigmaIntegratorLast2 <= sigmaIntegrator2;

if sigmaIntegrator2 >= 0 then
    sigmaOut := '1';
else
    sigmaOut := '0';
end if;

outputPin <= sigmaOut;

if sigmaOut = '1' then
    sigmaDac <= "00" & X"7FFFFF";
elsif sigmaOut = '0' then
    sigmaDac <= "11" & X"800001";
end if;

end if;
end process;

UpdateValue : process(clk)
begin
    if reset_n = '0' then
        last_data1 <= (others => '0');
    elsif falling_edge(clk) then
        if ast_sink_valid = '1' then
            -- get value from ST Bus
            last_data1 <= signed(ast_sink_data);
        end if;
    end if;
end process;

ClkDivider : process(clk)
variable counterDivider : unsigned(2 downto 0);
begin
    if reset_n = '0' then
        counterDivider := to_unsigned(0,3);
        ClkDividerOut <= '0';
    elsif rising_edge(clk) then
        if counterDivider = 4 then
            counterDivider := to_unsigned(0,3);
            -- clk out = 12MHz / 10 = 1,2MHz
            ClkDividerOut <= not ClkDividerOut;
        else
            counterDivider := counterDivider +
                to_unsigned(1,3);
        end if;
    end if;
end process;
end architecture;
```

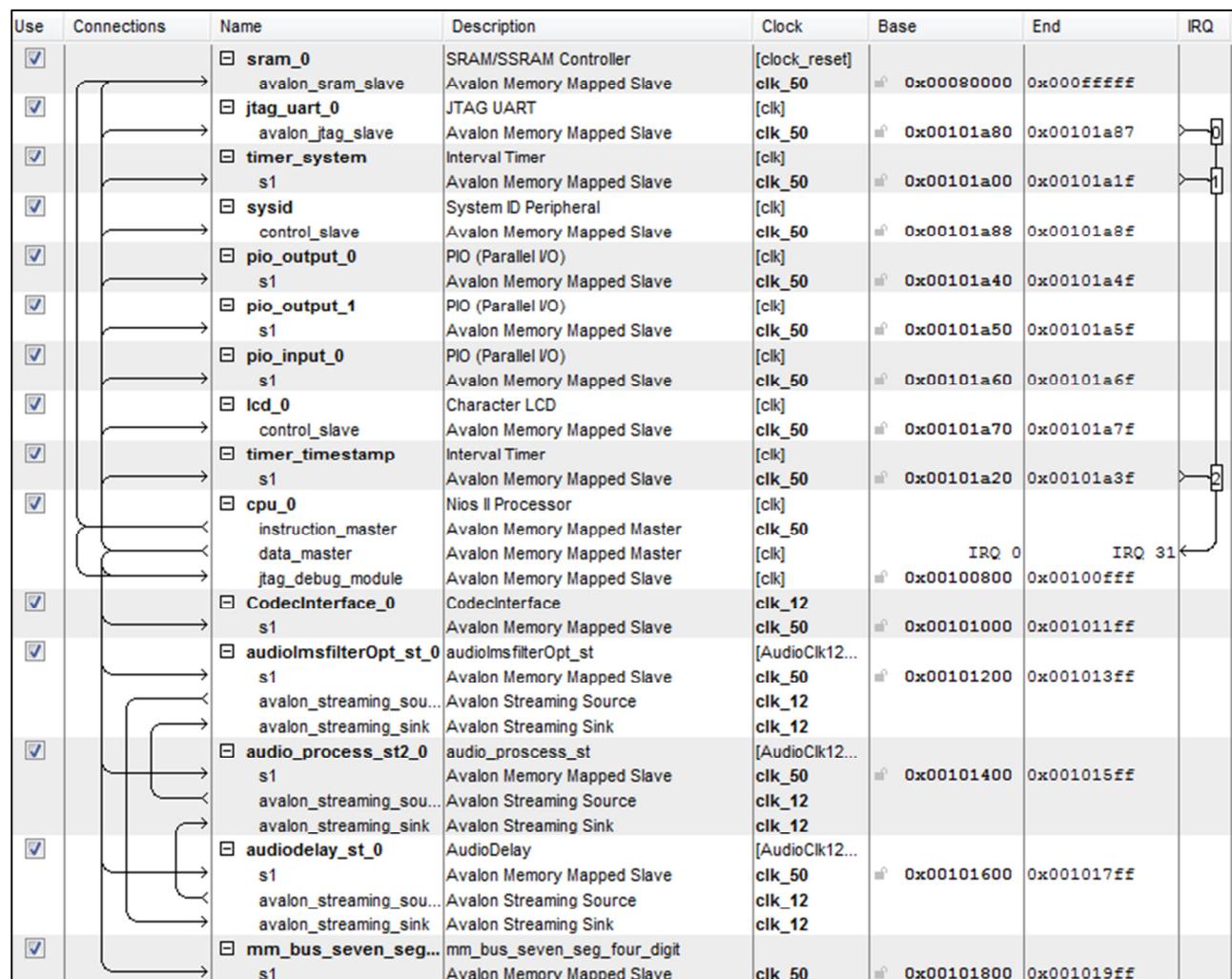
Kode 10 Sigma delta konverter med upsampler, FIR LP filer, og kvantisator (UpSampler.vdh)

6 SoPC systemer

I de følgende afsnit beskrives kort de 2 SoPC projekter vi har bygget, der benytter komponenterne beskrevet i de forgående kapitler. Det første projekt beskriver et SoPC system med Nios II processor med sigma delta konverter, der på 2 digitale udgange med et efterfølgende analogt filter afspiller stereo lyd. Det andet projekt indeholder stereo delay og LMS filter som kan styres fra Nios II processoren. Der er udviklet et simplet C-program til begge projekter, der kan demonstrere de komponenter vi har tilføjet SOPC projekterne.

6.1 SoPC - LMS filter med audio delay

Med altera SoPC builderen, har vi bygget et SoPC projekt, hvor vi har en række komponenter udviklet i faget ETDSPC samt audio delay og LMS filteret beskrevet i denne rapport. Figur 11 viser, hvilke og hvordan komponenterne er forbundet i SoPC builderen. ST komponenterne kan simpelt routes ved at forbinde source til sink ST interface, dette gør det nemt at bygge en ny applikation med forskellige lydbehandlings komponenter. Projektet indeholder også en række standard Altera komponenter samt en 7 segments driver (**mm_bus_seven_segment**), vi har udviklet i kurset.



Figur 11 SoPC Builder, der viser alle komponenterne der indgår i SoPC projektet

Figur 12 viser hvordan en PLL er indsatt, der fra 50 MHz krystallet på DE2 boardet laver en 50 MHz og 12 MHz clock, som er i fase med hinanden. Dermed sikres at vi arbejder med det samme clock domæne. I2S signalerne fra audio codec'en fra Wolfson Microelectronics (WM8731), er forbundet direkte til **CodecInterface** komponenten, som sender audio data på den synkrone audio bus til **audio_prosses_st2**, der konverterer audio kanalerne til formatet på ST bussen. **CodecInterface** komponenten har et I2C interface, der via hardwaren foretager opsætning af code'en. Rutning af den synkrone audio bus er udført manuelt (**audioout** -> **audioin**) som ses i Figur 12.



Figur 12 Forbindelser af SOPC_Systemet med DE2 boardet

Softwareen, der er skrevet til Nios II processoren, implementerer en simpel menu, hvor det er muligt at give kommandoer til ændring af hardware komponenternes funktioner i real-time.

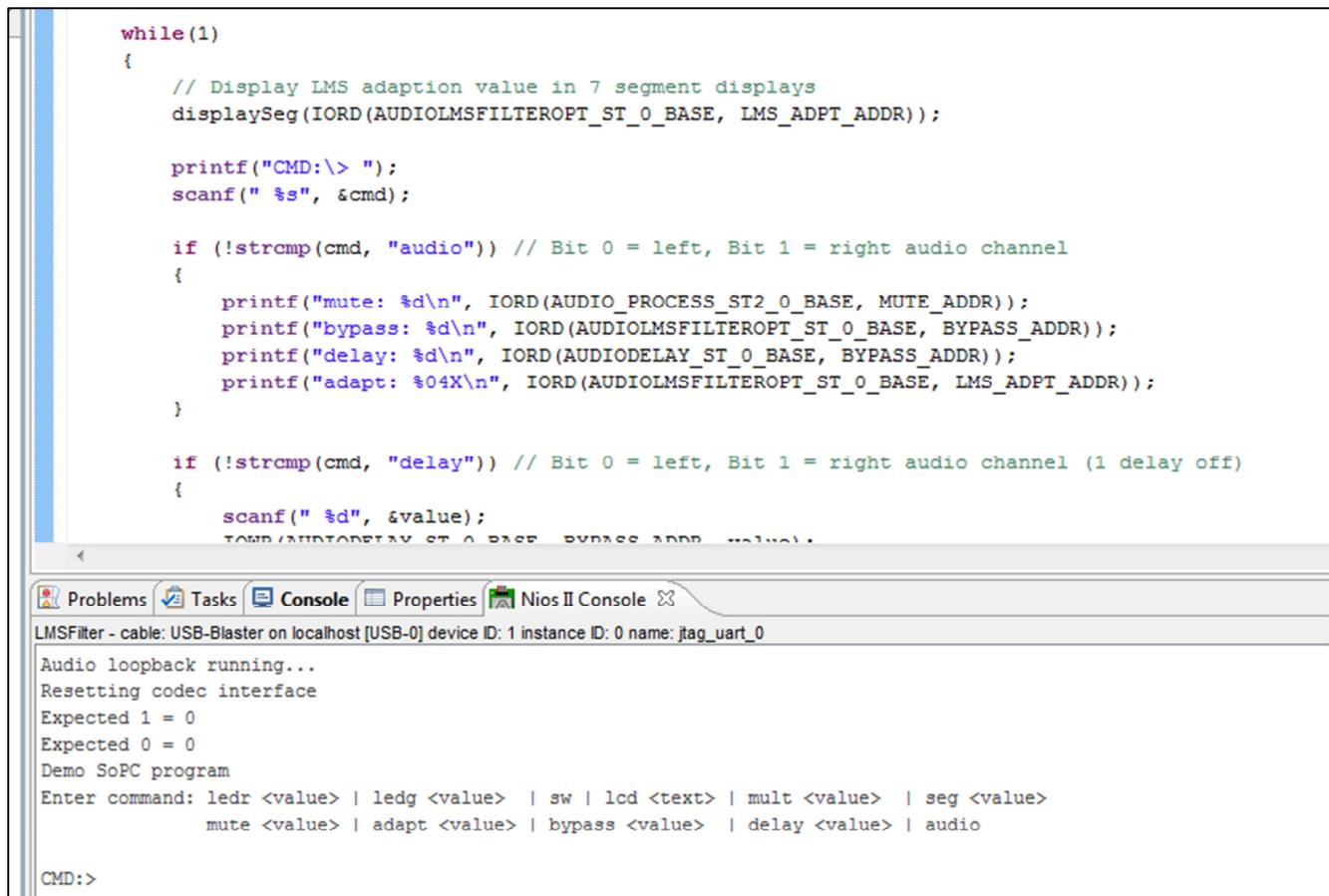
Programmet læser via MM bussen registrene i de forskellige komponenter, der er implementeret. Følgende kommandoer kan afgives via Nios II Consolen, som har forbindelse til NIOS processoren via JTAG Uart'en. Programmet findes i filen **mySopclMSFilter_SW.zip** se appendix B.

For audio komponenterne kan status aflæses for mute, bypass af LMS filter, bypass af audio delay og adaptions koefficienten. Disse værdier kan ændres med kommandoerne:

- **audio**, aflæser status for audio ST komponenterne
- **mute <value>**, mute af input bit 0 = venstre audio kanal og bit 1 = højre audio kanal
- **adapt <value>**, decimalt 24 bit fixed point værdi af adaptions konstant
- **bypass <value>**, bit 0 = LMS filter bypass venstre kanal, bit 1 = LMS filter bypass højre kanal
- **delay <value>**, bit 0 = Delay bypass venstre kanal, bit 1 = Delay bypass højre kanal

Komponenter til lydbehandling i en FPGA

De øvrige kommandoer styre funktioner fra øvelserne vi har implementeret i kurset, herunder implementeringen af en customized instruktion til optimering af en matrix multiplikation. (**mult 1 – SW version, mult 2 – HW version**)



The screenshot shows the Nios II software interface. The top part is a code editor with C code for audio processing. The bottom part is a terminal window showing the execution of the program.

```
while(1)
{
    // Display LMS adaption value in 7 segment displays
    displaySeg(IORD(AUDIOLMSFILTEROPT_ST_0_BASE, LMS_ADPT_ADDR));

    printf("CMD:> ");
    scanf(" %s", &cmd);

    if (!strcmp(cmd, "audio")) // Bit 0 = left, Bit 1 = right audio channel
    {
        printf("mute: %d\n", IORD(AUDIO_PROCESS_ST2_0_BASE, MUTE_ADDR));
        printf("bypass: %d\n", IORD(AUDIOLMSFILTEROPT_ST_0_BASE, BYPASS_ADDR));
        printf("delay: %d\n", IORD(AUDIODELAY_ST_0_BASE, BYPASS_ADDR));
        printf("adapt: %04X\n", IORD(AUDIOLMSFILTEROPT_ST_0_BASE, LMS_ADPT_ADDR));
    }

    if (!strcmp(cmd, "delay")) // Bit 0 = left, Bit 1 = right audio channel (1 delay off)
    {
        scanf(" %d", &value);
        TMRD /AUDIODELAY_ST_0_BASE_BYPASS_ADDR_value;
    }
}
```

LMSFilter - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtag_uart_0

```
Audio loopback running...
Resetting codec interface
Expected 1 = 0
Expected 0 = 0
Demo SoPC program
Enter command: ledr <value> | ledg <value> | sw | lcd <text> | mult <value> | seg <value>
               mute <value> | adapt <value> | bypass <value> | delay <value> | audio

CMD:>
```

Figur 13 Nios II software med console

Det færdige projekt bruger 22 % af FPGA ram memory, 29% af LE og 31% af de embedded multipliers. Den maksimale clock frekvens (Fmax) er 78.47 MHz (50 MHz) og 26.11 MHz (12 MHz) så vi er godt fra grænserne med en margin på næsten 100 %.

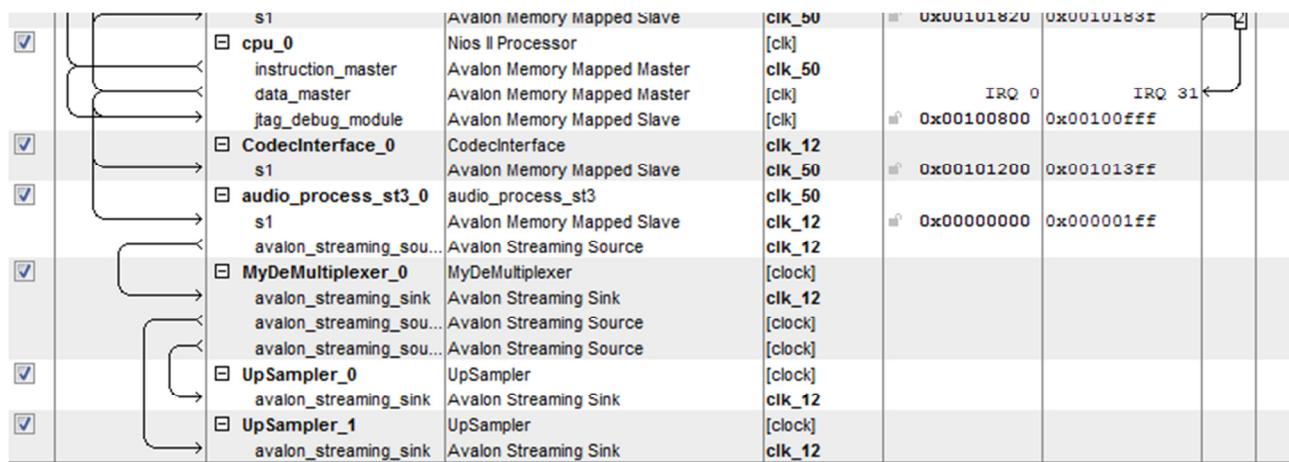
| | |
|------------------------------------|---|
| Flow Status | Successful - Thu Mar 15 14:01:06 2012 |
| Quartus II Version | 11.0 Build 157 04/27/2011 SJ Full Version |
| Revision Name | firstSopc |
| Top-level Entity Name | firstSopc |
| Family | Cyclone II |
| Device | EP2C35F672C6 |
| Timing Models | Final |
| Total logic elements | 9,645 / 33,216 (29 %) |
| Total combinational functions | 7,448 / 33,216 (22 %) |
| Dedicated logic registers | 6,385 / 33,216 (19 %) |
| Total registers | 6385 |
| Total pins | 114 / 475 (24 %) |
| Total virtual pins | 0 |
| Total memory bits | 108,416 / 483,840 (22 %) |
| Embedded Multiplier 9-bit elements | 22 / 70 (31 %) |
| Total PLLs | 1 / 4 (25 %) |

Figur 14 LMS filter og audio delay FPGA resource forbrug

6.2 SoPC - Sigma delta konverter

(Rune)

Dette afsnit er ikke beskrevet, kun illustreret med tegninger – der henvises til SoPC projektet appendix B.



Figur 15 Stereo sigma delta converter med demultiplexer I SOPC builder

Komponenter til lydbehandling i en FPGA



Figur 16 Forbindelser af SOPC systemet til DE2 boardet (GPIO_1 benyttes som pins)

| | |
|------------------------------------|---|
| Flow Status | Successful - Thu Mar 22 10:37:24 2012 |
| Quartus II Version | 11.0 Build 157 04/27/2011 SJ Full Version |
| Revision Name | firstSopc |
| Top-level Entity Name | firstSopc |
| Family | Cyclone II |
| Device | EP2C35F672C6 |
| Timing Models | Final |
| ▲ Total logic elements | 5,819 / 33,216 (18 %) |
| Total combinational functions | 4,213 / 33,216 (13 %) |
| Dedicated logic registers | 3,724 / 33,216 (11 %) |
| Total registers | 3724 |
| Total pins | 88 / 475 (19 %) |
| Total virtual pins | 0 |
| Total memory bits | 11,264 / 483,840 (2 %) |
| Embedded Multiplier 9-bit elements | 28 / 70 (40 %) |
| Total PLLs | 1 / 4 (25 %) |

Figur 17 Sigma delta converter FPGA resource forbrug

7 Resultater og diskussion

Vi har sammenfattet en oversigt over de udviklede komponenter til lydbehandling, med en oversigt over interface audio sync eller ST bus. Implementations style herunder pipelined eller state machine. Filterorden eller delay taps. FPGA ressourcer herunder LE, registeres, multipliers og memory. % FPGA viser, hvor mange ressourcer der benyttes af Cyclon II FPGA'en på DE2 boardet. Latency er angivet for en audio clock frekvens på 12 MHz, hvor en teoretisk værdi er beregnet ud fra antal stages i pipeline.

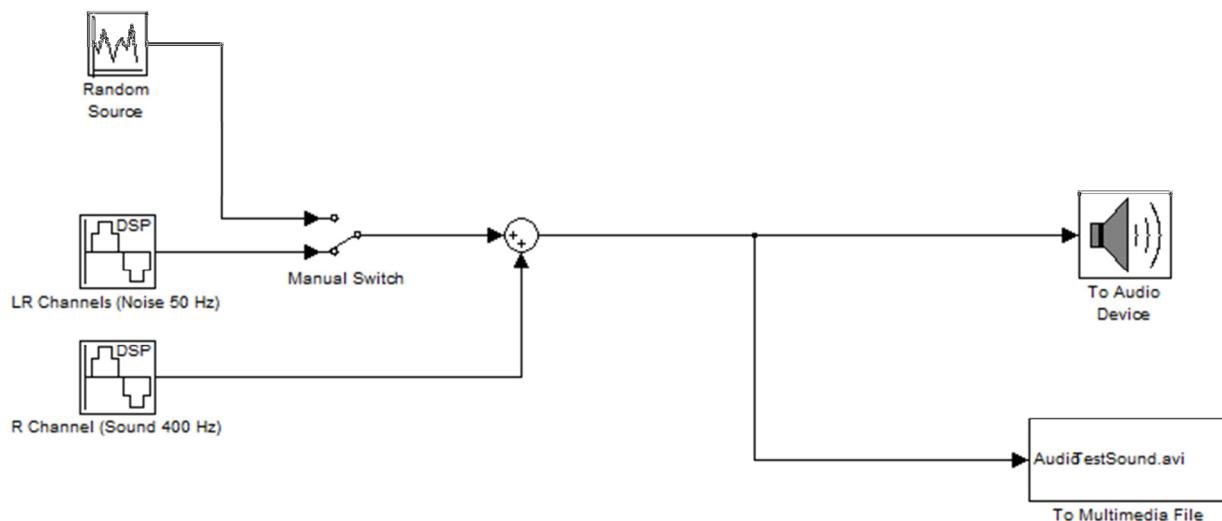
| Komponent | Interface | Style | Type | LE Registers Multipliers Memory (% FPGA) | Latency (12 MHz) | Restricted Fmax |
|--|-----------|-------------------------------|--------------------|--|---------------------|--------------------|
| LMS filter (speed) (audiolmsfilter) | Sync Bus | Pipelined 3 stages | 10 taps | 11354 876 70 0 (34 %) | 250 ns | 28.66 MHz |
| LMS filter (area) (audiolmsfilterOpt_st) | ST Bus | States 3xtaps+3 | 64 taps | 6832 4898 16 0 (21 %) | 16 us | 28.05 MHz |
| Audio delay (audiodelay_st) | ST Bus | Memory block | 2048 delay taps | 179 124 0 97152 (20 %) | 42 ms | 156.79 MHz |
| Sigma Delta converter (UpSampler) | ST Bus | UpSample FIR Integrator | 26 taps | 1623 1154 12 0 (17 %) | - | 54.76 MHz |
| FIR symetric (audiofilter) | Sync Bus | Pipelined 3 stages | 10 taps | 631 467 6 0 (2 %) | 250 ns | 87.4 MHz |
| FIR transposed (audiotransposedfilter) | Sync Bus | Pipelined 3 stages | 10 taps | 481 439 9 0 (1 %) | 250 ns | 194 MHz |
| IIR filter (audioIIR_st) | ST Bus | Pipelined 4 stages | Biquad | 918 679 35 0 (50 %) | 333 ns | 123 MHz |

Det bemærkes, at der benyttes få multipliers for FIR symetrisk og transposed filterne, det er fordi at filterne multipliceres med konstanter, hvormed compileren (syntesen) har mulighed for at optimere designet. Det kunne være at benytte distribueret aritmetik i look up tabeller se "Binary Adders, Chapter 2.3" i [4]. IIR filteret er meget hurtigt op til 123 Mhz på bekostning af mange multipliers. Det næste step ville være at

Komponenter til lydbehandling i en FPGA

implementere en version optimeret for area med beregning af filter koefficienterne fra software, hvilket versionen er forberedt til. Dette IIR filter ville være grundlaget for implementering af eksempelvis en grafisk equalizer, hvor betjening af denne kunne implementeres i software på Nios processoren. IIR filter strukturen benyttes i mange audio algoritmer, her kunne en yderligere optimering være aktuel med emner som optimering af multipliers se kapitel 2.3 [4].

Det endelige SoPC design med LMS filter og audio delay er testet ved at sende et uønsket sinus signal i den ene audio kanal og et ønsket moduleret med uønsket sinus signal i den anden kanal. Det var muligt at med fjerne et uønsket signal med kun 1 Hz i forskel mellem ønsket og uønsket signal med samme amplitude. Nedenstående viser hvordan teststimuli er generet med en Simulink model. Input er testet med 50-400 Hz. Med et test signal på 400 Hz (ønsket) og 399 Hz (uønsket), filtreres det uønskede 399 Hz signal væk.



Figur 18 Simulink model til test af LMS filter (Simulink_audiotest.mdl)

Audio delay funktionen er testet ved at afspille et stereo musik signal og sende kommandoen delay 0-3 fra Nios II konsollen se Figur 13. Se vedlagte video for en demo af projektet.

8 Konklusion

Projektet har givet os meget praktisk erfaring med implementation og optimering af signalbehandling algoritmer for en FPGA platform bestående af både digital hardware og softcore microprocessor. Vi har vist hvordan komponenter kan udvikles og genanvendes i et SoPC design med et simplificeret Avalon Streaming Interface. Vi har fundet en god metode for verifikation med ModelSim kombineret med reference modeller i MATLAB eller C-kode. Algoritmerne, som er implementeret kunne sagtens benyttes for systemer med en samplings rate op til 100 MHz. For lydbehandling (48-96 KHz) er der muligt med mange komponenter i serie på ST bussen og stadig nå filterberegningerne indenfor en sample. Det har været et både lærerigt og spændende projekt med et samlet forbrug på ca. 100 effektive projekttimer.

Det er ikke alle emnerne vi har formået at tage med i projektet, men vi har forsøgt at afdække mange af læringsmålene, som er opsummeret nedenstående liste.

Læringsmål:

- Implementere programmer for FPGA'er, skrevet i VHDL
 - o Er demonstreret i høj grad, med mange komponenter
- Anvende ModelSim og test benches til at udføre simulation af VHDL design
 - o Mange test benches er implementeret for verifikation af komponenterne
 - o Vi har ikke benyttet assertions, men i stedet anvendt ModelSim's timings diagrammer og opsamling af data i tekst filer for verifikation med MATLAB modeller
- Anvende constraints til specifikation af system krav
 - o Der er krav til systemet, om samplingsfrekvens og systemets forskellige clocks
- Redegøre for begreber som: clock domæner, clock skew, pipelining, PLL- og memory komponenter
 - o Vi arbejder et clock domæne, bestående af en 50 MHz og 12 MHz clock genereret af en PLL, der sikrer samme fase
 - o Der er anvendt memory komponenter til implementering af en audio delay
 - o Vi har ikke behandlet emnet clock skew eller problemer som metastabilitet i forbindelse med "clock domain crossing issues" [7]
- Redegøre for timings simulering og analyse i Quartus II værktøjet
 - o Kun funktionel simulering med ModelSim er udført
 - o Timings analyse med Quartus II er foretaget til bestemmelse af Fmax
- Anvende soft cores til opbygning af et SoC (System On Chip) system
 - o To forskellige SoPC systemer er implementeret og demonstreret
- Implementere C programmer til afvikling på SoC
 - o Et program til ændringer af komponenternes opsætning
- Implementere signal behandlings algoritmer i VHDL
 - o FIR filter typer, LMS filter, Sigma Delta konverter og IIR filter

9 Appendix A - Modeller

Dette appendix indeholder MATLAB og C-koden for modellerne af LMS filteret og sigma delta konverter.

9.1 MATLAB - LMS filter

```
%%%%%%  
%  
% LMSFilter  
%  
% By Kim Bjerge IHA  
%  
%%%%%%%%%%  
function [y,e] = LMSFilter(x,d,N,u)  
%% Adaptive filtering using LMS  
% Adaptive filtering of the input vector x using the desired vector d  
% The algorithm performs an adaptive FIR filtering on the input x  
% The LMS algorithm updates the filter weights/coefficients  
% according to:  
%   y(n) = w(n)*x(n) = sum{w(l)x(n-l)}    (FIR filter for l=1:N)  
%   e(n) = d(n)-y(n)                         (Estimation error)  
%   w(k;n) = w(k;n-1)+f(k;x(n),e(n),u)      (LMS update of weights)  
% where  
%   f(k;x(n),e(n),x) = ue(n)x*(n-k)  
%  
% Complex conjugate of the input vector, assuming only real value of x  
%  
% Parameters:  
%  
% x - input vector (Must be normalized with peak(x) < 1)  
% w - weights (FIR filter coefficients)  
% u - adaptation step size  
% N - vector size of filter weights  
%  
% y - output vector  
% e - estimation error  
  
% Initialize vectors  
w = zeros(1,N);          % Weights w = 0  
y = zeros(1,length(x)); % Output vector  
e = zeros(1,length(x)); % Estimation error  
  
for n=1:length(x)  
  
    % FIR filter input vector  
    for l=1:N  
        if (n > l)  
            y(n) = y(n) + w(l)*x(n-l+1);  
        end  
    end  
  
    % Estimate error  
    e(n) = d(n) - y(n);
```

```
% Adjust weights
for k=1:N
    if (n > k)
        w(k) = w(k) + u*e(n)*x(n-k);
    end
end

%% For debug only - final weights filter plot
xn = [0:0.03:pi];
HH = freqz(w, 1, xn);
xn = xn.* (500/(2*pi));

figure(3);
subplot(2,1,1);
plot(xn, abs(HH));
title('Amplitude response of LMS filter');
xlabel('frequence');
ylabel('amplitude');
subplot(2,1,2);
plot(xn, angle(HH));
title('Phase response of LMS filter');
xlabel('frequence');
ylabel('phase');
```

```
%%%%%
%
% LMSNoiseSupressionSolution
%
% By Kim Bjerge IHA
% Implementation of LMS filter
%
%%%%%
f1 = 8000; % Undesired signal
f2 = 800;
f3 = 2000;
fs = 48000; % signal parameters
N = 1000; n = 0:1:N-1; % length and time index

sn1 = sin(2*pi*f1*n/fs); % generate sinewave 8000 hz
sn2 = sin(2*pi*f2*n/fs); % generate sinewave 800 hz
sn3 = sin(2*pi*f3*n/fs); % generate sinewave 1000 hz

%noise=randn(size(sn1)); % generate random noise
noise = wgn(N,1,0)'; % generate random noise
dn = 0.3*sn1+0.3*sn2+0.3*sn3+0.01*noise; % mixing desired sinewaves
xn = 0.4*sn1+0.1*noise; % generate x(n) with noise and undesired signal

L = 64; % filter length
mu = 0.004; % step size mu

[y,e] = LMSFilter(xn,dn,L,mu); % LMS Filter function
```

```
figure(1);
plot(dn);
title('Noisy input signal');

figure(2);
title('Frequency response of input');
freqz(dn);

figure(4);
plot(e, 'b');
title('Adaptive filter');
xlabel('Time index, n'); ylabel('Amplitude');

figure(5);
load 'rightoutlms.txt'
vhdl = rightoutlms./(2^23);
hold on
plot(e, 'r');
plot(vhdl, 'b');
hold off
title('VHDL (blue) vs. MATLAB (red)');
xlabel('Time index, n'); ylabel('Amplitude');

figure(6);
title('Frequency response of output');
freqz(e);

SaveAsFixedInFile(xn, 'Noise.txt');
SaveAsFixedInFile(dn, 'NoiseSignal.txt');
```

9.2 C-Kode LMS filter

```
// Initialization of LMS filter setting delayLine and coefficients buffer
void initLMSFilter(short *delayLine, short *weights, short length, short
stepSize)
{
    short k;

    delay = delayLine;
    w = weights;
    L = length;
    u = stepSize;

    // Clear weights and delay line
    for (k = 0; k < L; k++)
    {
        w[k] = 0;
        delay[k] = 0;
    }
}

void LMSFilter(short x, short d, short *y, short *e)
{
```



```
int yn=0, wk_i;
short k;
short out, err, wk_s;

short len = L;                                // For optimization
short *dly = delay;                          // For optimization
short *wgt = w;                            // For optimization
short adpt = u;                           // For optimization

// Shift delay line
for(k=len-1; k > 0; k--)
    dly[k] = dly[k-1];

// Insert next x
dly[0] = x;

// Convolution: w * x
for(k=0; k < len; k++)
    yn += wgt[k] * dly[k];

// Calculate output result
out = (yn >> 15);

// Estimate error (n)
err = d - out;

// Adjust weights
for(k=0; k < len; k++)
{
    wk_i = err*dly[k];
    wk_s = (wk_i >> 15); // Truncate
    wk_i = adpt*wk_s;
    wgt[k] += (wk_i >> 15); // Truncate
}

// Return output and error estimate
*y = out;
*e = err;
}
```

```
#define NUM_SAMPLES 1000

// LMS Filter section
short LMSDelay[LMSLen];
short LMSWeights[LMSLen];

short d[NUM_SAMPLES] = {
    "NoiseSignal.txt"
};  

#include

short x[NUM_SAMPLES]= {
    "Noise.txt"
};  

#include

short error[NUM_SAMPLES];

void initAdaptiveFilter(void)
```

```
{  
    initLMSFilter(LMSDelay, LMSWeights, LMSLen, AdaptationStepSize);  
}  
  
int main(void) {  
    int n;  
    short output;  
    FILE *fp_out; // Output test file  
  
    puts("Fixed Point version of LMS filter");  
    fp_out = fopen("OutputSignal.txt", "w");  
  
    initAdaptiveFilter();  
  
    for (n = 0; n < NUM_SAMPLES; n++)  
    {  
        LMSFilter(x[n], d[n], &output, &error[n]);  
        fprintf(fp_out, "%d\n", error[n]);  
    }  
  
    fclose(fp_out);  
    puts("Filtered output signal created : OutputSignal.txt");  
  
    return EXIT_SUCCESS;  
}
```

9.3 MATLAB – Sigma Delta Converter

```
%%%%%%%%%%%%%  
%  
% LMSFilter  
%  
% By Kim Bjerge IHA, modified by Rune Salberg-Bak  
%  
%%%%%%%%%%%%%  
clear all;  
  
% vælg mellem Sigma Delta eller 1-bit kvantisator  
SIGMA_DELTA = 1;  
  
% samples i input signal  
N = 100;  
  
% oversamplings-rate 48 KHZ * 256  
M = 25;  
%M = 256;  
  
% Lavpas filter med knæk ved: (fs/2)/M  
%filtLen = 300;  
filtLen = 26;  
n = (-filtLen/2filtLen/2);  
h = (1/M)*sinc(n/M) .* hann(filtLen+1)';  
% normalisere h(n) så den har et areal på 1  
h = h/sum(h);  
H = abs(fft(h,N*M));
```

```
% gem i fil
h16 = h*1;
figure(1)
freqz(h16);

figure(2)
h16 = round(h*512*2);
freqz(h16);
fid = fopen('coeff_mat.txt', 'w');
for i=1:length(h16)
    xtext = num2str(h16(i));
    fprintf(fid, '%s,', xtext);
end
fclose(fid);

% Modified for Matlab 7
fid = fopen('coeff.txt', 'w');
for i=1:length(h16)
    xtext = num2str(h16(i));
    fprintf(fid, '%s,', xtext);
end
fclose(fid);

% Inputtet bevæger sig i intervallet -1..1
x = 0.9*sin(2*pi*(0:N)/N);
x16 = x*2^15;

% Original version
xtxt = [num2str(x16(1))];
for i=2:length(x16)
    xtxt = [xtxt ','];
    xtxt = [xtxt num2str(x16(i))];
end
save 'x_mat.txt' xtxt;

% Modified for Matlab 7
fid = fopen('x.txt', 'w');
for i=1:length(x16)
    xtext = num2str(x16(i));
    fprintf(fid, '%s,', xtext);
end
fclose(fid);

% Up-sample
for n=1:N
    for m=1:M
        xM(m + (n-1)*M) = x(n);
    end
end

% Lavpas filtrere
xMLP = filter(h,1,xM(1:M*N));

% init
y1 = 0;
xio = 0;
```

```
xio2 = 0;
last = 0;

if SIGMA_DELTA == 1
    disp('Sigma_Delta konverter');
    %Sigma-Delta konverter
    for n=1:N*M
        xi = xMLP(n) - y1;
        xio = xi + xio;

        %xio2 = xio - y1;
        xio2 = xio2 + xio - y1;

        %quantize
        if xio2>=0
            y1 = 1;
        else
            y1 = -1;
        end

        y(n) = y1;
    end
else
    disp('1-bit kvantisator - dvs. ingen noise-shaping');
    % 1-bit kvantisator - dvs. ingen noise-shaping
    for n=1:N*M
        % quantize
        if xMLP(n)>=0
            y1 = 1;
        else
            y1 = -1;
        end

        y(n) = y1;
    end
end

% Flyt xMLP i tiden så den passer med yLP
% OBS: Dette filter har ingen praktisk betydning, det er indsat alene for at
skaleringen passer!
xMLP = filter(h,1,xMLP);

% fejl
e = y - xMLP(1:N*M);

% filtrer y
yLP = filter(h,1,y);

% fjerner fejlen for de første (filtLen*2) --Rune
for n=1:(filtLen*2)
    yLP(n) = xMLP(n);
end

% fejl i det lavpas filtrerede signal
eLP = xMLP - yLP;
```

Komponenter til lydbehandling i en FPGA

```
disp(['Varians af input signal x: ' num2str(var(xM))]);
disp(['Fejl-varians af det lavpasfiltrede signal: ' num2str(var(eLP))]);
disp(['Praktisk SNR: ' num2str(10*log10(var(xM)/var(eLP))))]);
A = 1;
var_Q = A^2/3;
phi_QLP = 2*var_Q/M - (2*var_Q/pi)*sin(pi/M) % Denne beregning forstår jeg ikke
(var_Q og /pi) !!!
ps_Q = 2*pi/M - 2*sin(pi/M)
disp(['Teoretisk SNR: ' num2str(10*log10(var(xM)/phi_QLP))]);

figure(1)
plot(yLP);
hold on;
plot(xMLP, 'r');
hold off;
title('Lavpasfiltrede output fra delta-sigma konverter');
xlabel('samples');
ylabel('amplitude');
legend('y lavpas filterede', 'x');

figure(2)
E = (abs(fft(e))/(N*M)).^2;
plot(E(1:N*M/2));
hold on;
% Skalering af amplitude karakteristikken og Hq
plot(0.0005*(2*(1-cos(pi.*(0:1/(N*M/2):1)))), 'r');
plot(0.005*H(1:N*M/2), 'g');
hold off;
title('Effekttæthedsspektrum af Kvantiseringsfejl');
xlabel('vinkelfrekvens');
ylabel('amplitude^2');
legend('Kvantiseringsfejl i y', 'Hq overføringsfunktion', 'LP filter');

figure(3)
plot(eLP);
```

10 Appendix B – VHDL og SOPC oversigt

10.1 VHDL kode med test benches

Dette appendix indeholder en oversigt over VHDL koden for komponenterne og tilhørende test bench inkluderet i **VHDLCode.zip** filern. Der er flere version af komponenter på listen, som ikke er beskrevet i detaljer i denne rapport.

Audio komponenter og test bench for sync bus:

- **audio_process.vhd** : Audio process interface til sync bus
- **audio_process_tb.vhd** : Test bench
- **audiofilter.vhd** : Symetrisk FIR Filter med sync bus interface
- **audiofilter_tb.vhd** : Test bench
- **audiotransposedfilter.vhd**: Transposed FIR Filter med sync bus interface
- **audiotransposedfilter.vhd**: Test bench
- **audiolmsfilter.vhd** : LMS filter med sync bus interface
- **audiolmsfilter_tb.vhd**: Test bench

Komponenter til ST Bus Version 2:

- **audio_process_st2.vhd**: Audio process med ST bus interface
- **audio_process_st2_tb.vhd**: Test bench
- **audiolmsfilterOpt_st.vhd**: LMS filter optimeret med ST bus interface
- **audiolmsfilterOpt_st_tb.vhd**: Test bench
- **audiodelay_st.vhd, delay_ram.vhd**: Audio delay med ST bus interface
- **audiodelay_st_tb.vhd**: Test bench
- **audiolIR_st.vhd**: IIR filter, som kun er testet i ModelSim (Ikke beskrevet i rapporten)
- **audiolIR_st_tb**: Test bench

Komponenter til ST Bus Version 3:

(Rune)

- **audio_process_st3.vhd**: Audio Process til ST bus
- **MyDeMultiplexer.vhd**: ST Bus multiplexer for stereo
- **MyDeMultiplexer_tb.vhd**: Test bench
- **UpSampler.vhd**: Upsampler med 1. bits kvantisator

Test bench, hjælpe pakker og filer:

- **io_util.vhd**: læsning og skrivning af tekst filer
- **txt_util.vhd**: håndtering af tekst
- **NoiseHex.txt, NoiseSignalHex.txt**: Tekst filer med input samples i Hex format
- **Leftoutlms.txt, rightoutlms.txt**: Tekst filer med output samples fra LMS filter

10.2 SOPC projekt arkiver

I dette appendix er listet de SOPC arkivet med tilhørende software som er inkluderet i **SOPCProjects.zip** filen:

SOPC projekt arkiv for delay og LMS filter:

- **firstSopc_0903_LMSDelayFilter.qar**: Quartus II Archive File (Version 11.0)
- **mySopcLMSFilter_SW.zip**: Nios II software project archive (Version 11.0)

SOPC projekt arkiv for sigma delta konverter:

(Rune)

- **firstSopc_1103_SigmaDeltaConverter.qar**: Quartus II Archive File (Version 11.0)
- **mySopcSigmaDelta_SW.zip**: Nios II software project archive (Version 11.0)

11 Referencer

- [1] Woon-seng gan, Embedded Signal Processing with the Micro Signal Architecture, Wiley
- [2] Altera, Quartus II Handbook Version 11.1, chapter 11. Recommended HDL Coding styles,
http://www.altera.com/literature/hb/qts/qts_qii51007.pdf
- [3] Steve Kilts, Avanced FPGA Design, Wiley-Interscience 2007
- [4] Uwe Meyer-Baese, Digital Signal Processing with Field Programmable Gate Arrays, Springer 2001
- [5] Altera's hjemmeside : www.altera.com
- [6] Avalon Interface Specification:
http://www.altera.com/literature/manual/mnl_avalon_spec.pdf
- [7] Saurabh Verma and Ashima S. Dabare, Understanding clock domain crossing issues, December 2007