|  |
| --- |
|  |
| **Komponenter til lydbehandling i en FPGA** |
| **ETDSPC Projekt** |
|  |
| **Rune Salberg-Bak (08935), Kim Bjerge (20097553)** |
|  |
| **Q1 2012** |

Indhold

[1 Indledning 3](#_Toc320104080)

[2 Problemformulering 3](#_Toc320104081)

[3 Udviklingsstrategi 5](#_Toc320104082)

[4 Arkitektur 6](#_Toc320104083)

[5 Komponent beskrivelser og validering 12](#_Toc320104084)

[5.1 ModelSim Testbench 12](#_Toc320104085)

[5.2 FIR filter 15](#_Toc320104086)

[5.3 Sigma delta converter 18](#_Toc320104087)

[5.4 LMS filter 19](#_Toc320104088)

[5.4.1 FPGA Implementering 21](#_Toc320104089)

[5.5 Delay 27](#_Toc320104090)

[6 SoPC systemerne 30](#_Toc320104091)

[6.1 Sigma delta SoPC 30](#_Toc320104092)

[6.2 LMS filter SoPC 30](#_Toc320104093)

[7 Test resultater 34](#_Toc320104094)

[8 Konklusion 36](#_Toc320104095)

[9 Appendix A - Modeller 37](#_Toc320104096)

[9.1 MATLAB – LMS filter 37](#_Toc320104097)

[9.2 C-Kode LMS filter 39](#_Toc320104098)

[9.3 MATLAB – Sigma Delta Converter 41](#_Toc320104099)

[10 Appendix B – VHDL og SOPC overblik 45](#_Toc320104100)

[10.1 VHDL kode oversigt 45](#_Toc320104101)

[10.2 SOPC projekt arkiver 46](#_Toc320104102)

[11 References 46](#_Toc320104103)

# 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 Interface (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 kompoenterne 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 arbejdsdag 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 medtage i rapporten.

# 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. 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 Bus (MM) [6]. Audio komponenterne udvikles og testes med simulering i ModelSim. Komponenter skal være udviklet så de i princippet 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 generer 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 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

# 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 opløsning herunder fixed-point format og input/output format. I dette tilfælde skal input samples med en opløsning 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.

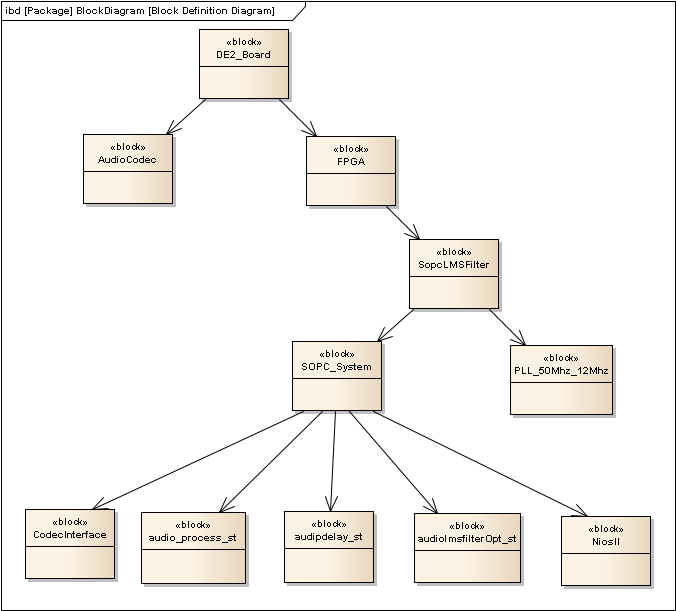
I projektet har vi fokuseret på at implementere ovenstående modeller i VHDL med DE2 boardet om target platform. Strategien er, at implementere en version af algoritmen i VHDL, som først simuleres og aftestes med en ModelSim test bench. Simuleringen benytter input test data produceret af modellerne fra 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.

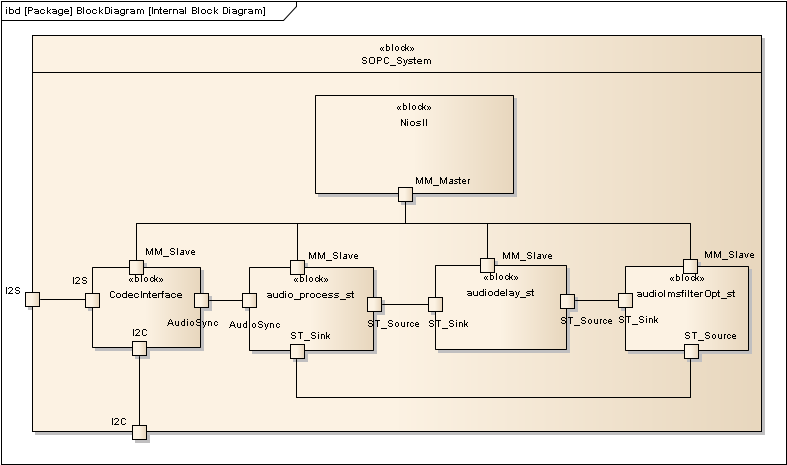
# Arkitektur

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, som er det centrale komponent for at interface FPGA’en med digital audio. 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. Komponenten **audio\_process\_st** konverterer et audio sync format fra **CodecInterface** til ST format. Herudover skal komponenterne **audiodelay\_st** og **audiolmsfilterOpt\_st** samt software til Nios II processoren udvikles til dette projekt. Der er indsat en PLL til generering af et 50 MHz og 12 MHz clock signal, der herved altid vil være i fase og dermed arbejder i samme clock domæne.



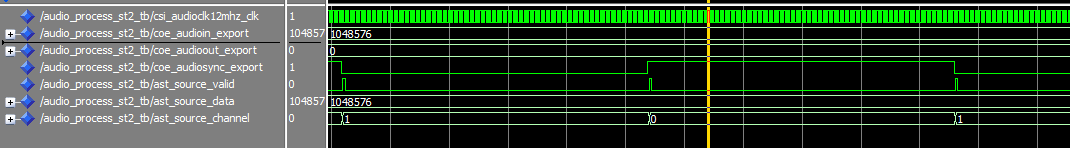
Figur 1 SysML arkitektur diagram

Nedenstående diagram viser, hvordan audio komponenterne er forbundet på henholdsvis Avalon Memory Management (MM) Bus og Streaming (ST) Interface, hvor audio delay og LMS filteret er indsat sammen med konvertering af audio sync til ST interface.



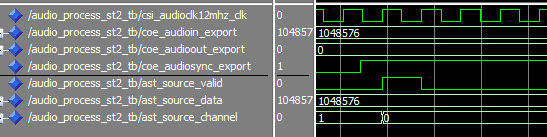
Figur 2 SysML internt arkitektur diagram for SOPC Systemet til LMS filter og audio delay

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



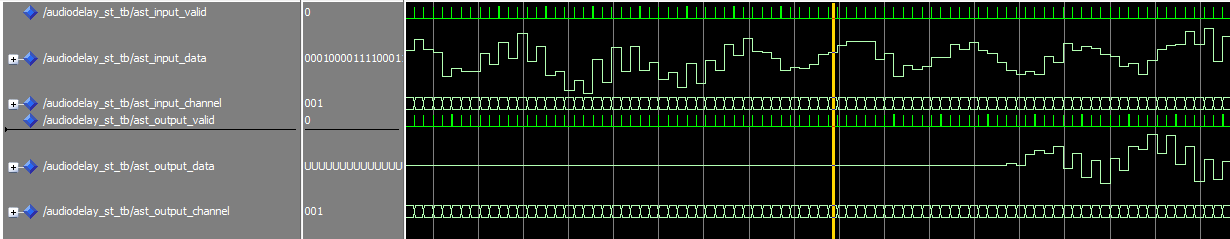
Figur 3 Audio sync til ST bus interface

Vi har valgt kun at benytte nedenstående signaler fra ST bussen (valid, data og channel). Channel angiver et nummer på den kanal, hvortil data tilhører. 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, dette har dog senere vist at være en dårlig idé, hvis komponenterne skal benyttes sammen med nogle af Altera’s Streaming komponenter.



Figur 4 ST bus timing for valid signal

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



Figur 5 Audio delay indsat 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. Her ses, hvilke forbindelser der er til audio interface, ST og MM bus.

**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** 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.vdh)

# Komponent beskrivelser og validering

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.

## ModelSim Testbench

Dette kapitel beskriver kort den sekventielle testbench, der er skrevet for at test audio komponenterne omfattende ST bus interfacetmed enten LMS filter, audio delay eller SigmaDeltaConverter. 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 **CodecInterface** og læser data 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, hvor indholdet kan sammenlignes med den ”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.vdh)

## FIR filter

Dette kapitel giver en kort introduktion til de forskellige FIR filter typer, vi har implementeret og testet med interface til sync bus 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 filtrer 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; -- see excel sheet

*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

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;

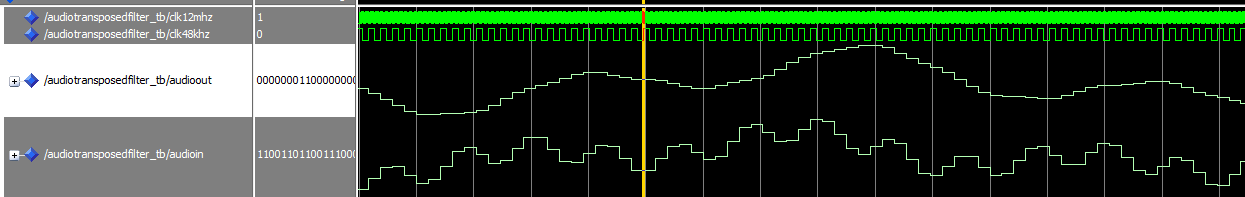
**end** **if**;

**end** **process** sample\_buf\_pro;

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

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 nemt afvikles med en audio clock på op til de 12 MHz.

Det symetriske filter bruger ca. halvt så mange multiplikationer og additioner som direkte form 1. Det tansposed 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 FIR filtered data

## Sigma delta converter

(Rune)

Dette afsnit er ikke bekrevet – der henvises til MATLAB koden i appendix A samt VHDL koden i appendix B.

## 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.

d(n)

+

x(n) y(n) -

Digital FIR filter

e(n)

Adaptiv LMS algorithm

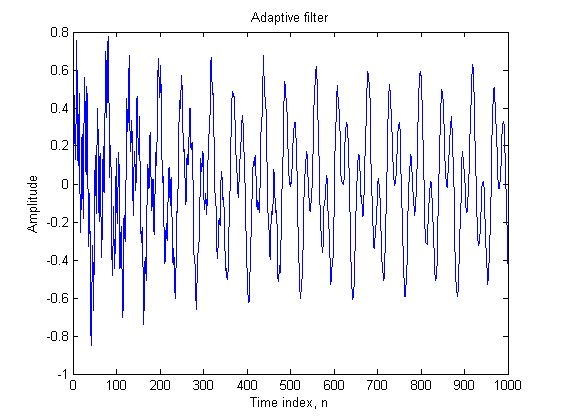
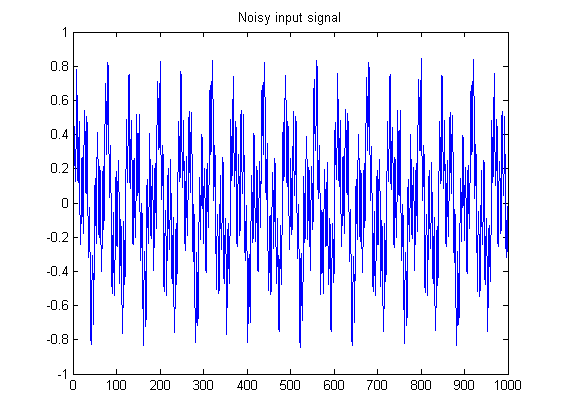
Figure 1 Adaptiv LMS filter

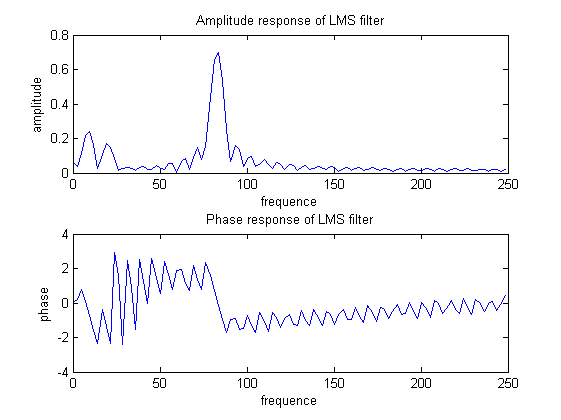
Det digitale FIR filter beregnes som

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

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

Ønsket er at implementerer et LMS filteret med DE2 boardet, hvor venstre audio kanalen indeholder et uønsket 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: **LMSNoiseSupressionSolution.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 filtrede 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.

### FPGA Implementering

Dette afsnit omfatter hvordan LMS filtreret 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 cyckler 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 opløsning 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'*);

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.vdh)

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 at 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 kører 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 resourcer 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 2x64 + 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**;

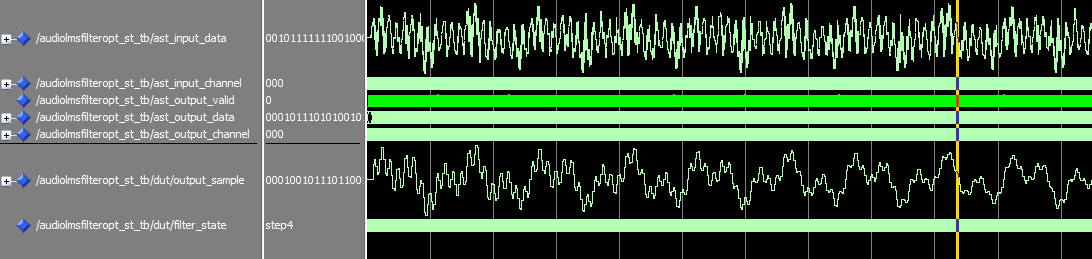
output\_sample <= error;

**end** **if**;

**end** **process** LMSFilter;

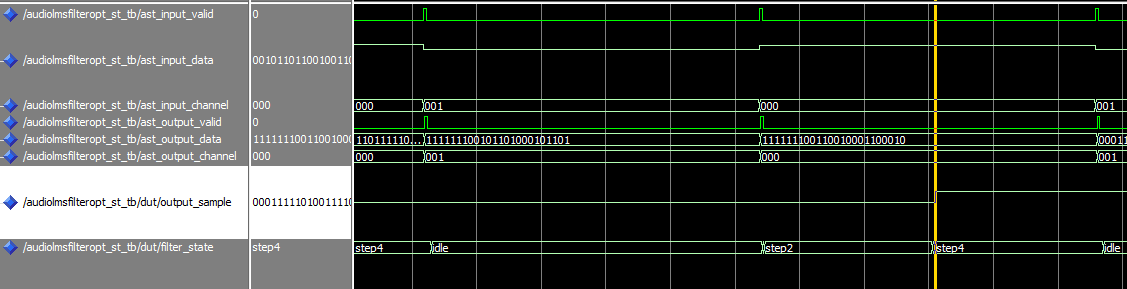
Kode 7 VHDL LMS filter med ST interface optimeret for area (audiolmsfilterOpt\_st.vdh)

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 timings simuleringer i ModelSim, hvor det kan ses på **output\_sample** signalet at LMS filteret justeret sig på plads.



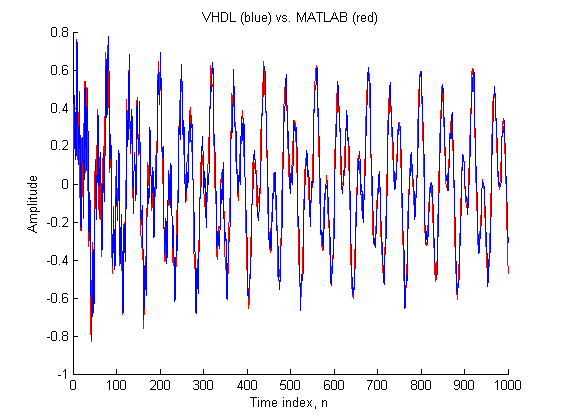
Figur 8 Verifikation af optimeret LMS filter med ST bus interface

Figur 10 viser timing for, hvordan valid signalere 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 elemeter 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 gængæld et meget hurtigt filter, der har et throughput på 24 bit \* 12 MHz = 288 Mbit/sec, med en latency på 166 ns. Teoretisk er det maksimale throughput på 24 bit \* 28.66 MHz = 687 Mbit/sec.

## 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], beskrives 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.vdh)

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 valg 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 := 2024;

*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**;

-- 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.vdh)

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 |

# SoPC systemerne

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.

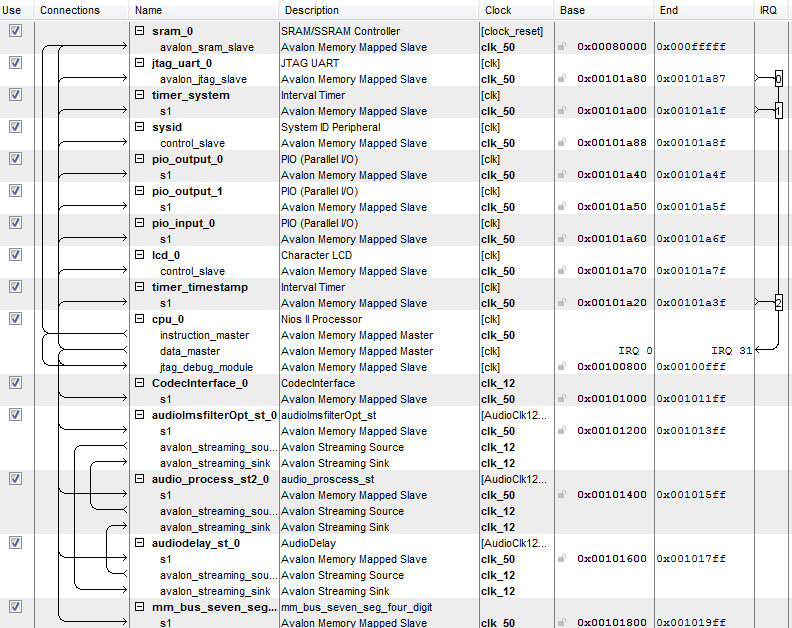
## Sigma delta SoPC

(Rune)

Dette afsnit er ikke bekrevet – der henvises til SoPC projektet appendix B.

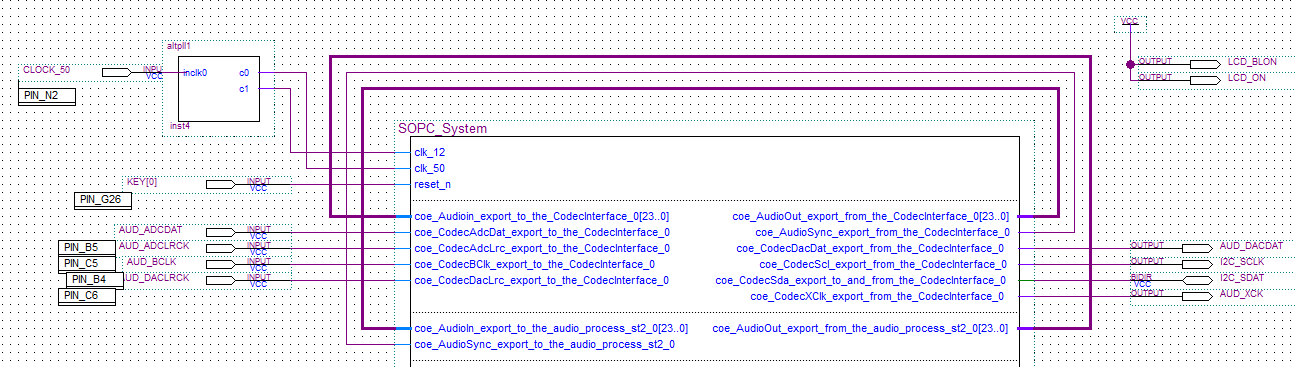
## LMS filter SoPC

Med altera SoPC builderen har vi bygget et SoPC projekt, hvor vi har en række komponenter udviklet i faget ETDSPC samt delay og LMS filteret beskrevet i denne rapport. Nedenfor er vist hvilke og hvordan komponenterne er forbundet i SoPC builderen. ST komponenterne kan simplet routes ved at forbinde source til sink interface. Projektet indeholder også en række standard Altera komponenter samt en 7 segments driver vi har udviklet i kurset.



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

Nedenfor er vist hvordan en PLL er indsat 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 domaine. 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\_prossses\_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 som ses i nedenstående figur.



Figur 12 Forbindelser af SOPC\_Systemet med DE2 boardet

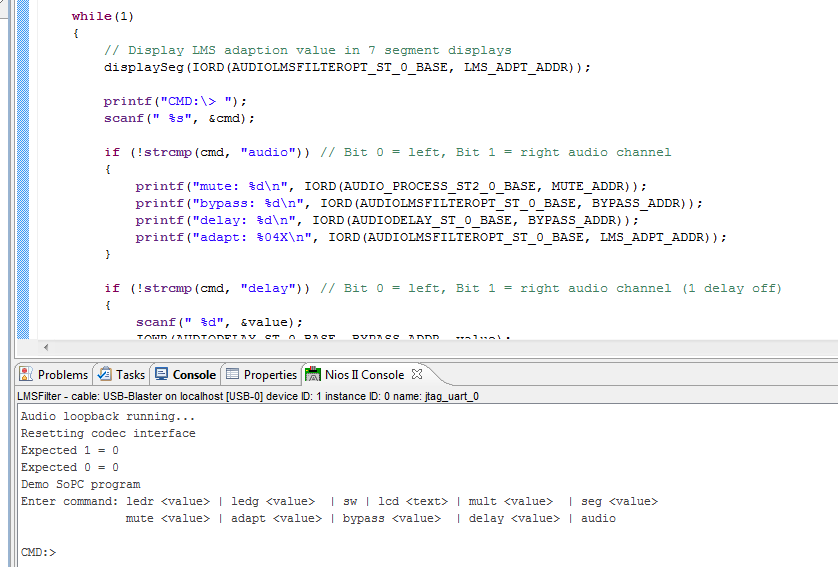
Softwaren, der er skrevet til Nios II processoren, implementere en simpel menu, hvor det er muligt at give kommandoer til ændring af hardware komponenternes funktioner i real-time.

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.

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:

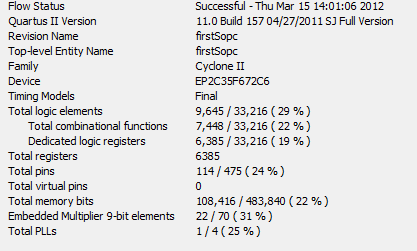
* **audio**, aflæser status for audio ST komponenterne
* **mute** <value>, mute af input bit 0 = venstre audio kanal , bit 1 = højre audio kanal
* **adapt** <value>, decimalt 24 bit fixed point værdi af adaptions konstant
* **bypass** <value>, bit 0 = LMS filter bypass venstre kanal, bit 1 = LMS filter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

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)



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 %.



Figur 14 FPGA resource forbrug

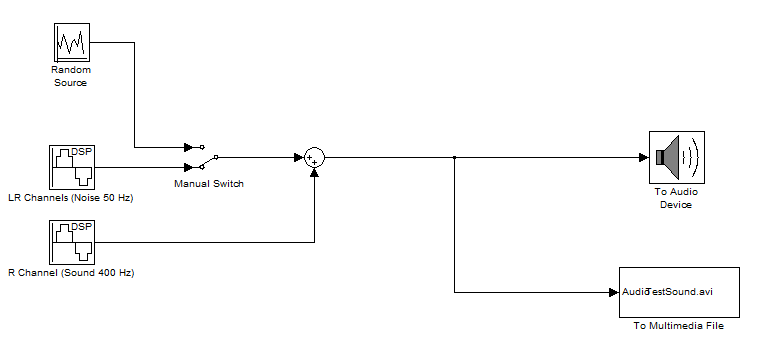
# Test resultater

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

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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 har mulighed for at optimere designet med automatisk at benytte distribueret aritmetik i look up tabeller se ”Binary Adders, Chaper 2.3” in [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 sinus signal i den anden kanal. Det var muligt at med fjerne et uønsket signal med kun 1 Hz i forskel med samme amplitude. Nedenstående viser vores teststimuli generet med en Simulink model. Input er testet med 400 Hz og 399 Hz, hvor de 399 Hz filteres fra.



Figur 14 Simulink model til test af LMS filter (Simulink\_audiotest.mdl)

# 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 embedded 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 verification med ModelSim kombineret med reference modeller i MATLAB eller C-kode. Algorithmerne som er implementeret kunne sagtens benyttes for systemer med en samplings rate op til 100 MHz. For lydbehandling (48 KHz) er der muligt med mange komponenter i serie på ST bussen.

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 nedenfor.

**Opsætning af mål:**

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

# Appendix A - Modeller

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

## MATLAB – LMS filter

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%

% LMSFilter

%

% By Kim Bjerge IHA

%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [y,e] = LMSFilter(x,d,N,u)

%% Adaptive filtering using LMS

% Aaptive 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

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('Frequence 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('Frequence response of output');

freqz(e);

SaveAsFixedInFile(xn, 'Noise.txt');

SaveAsFixedInFile(dn, 'NoiseSignal.txt');

## 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] = {

**#include** "NoiseSignal.txt"

};

**short** x[NUM\_SAMPLES]= {

**#include** "Noise.txt"

};

**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;

}

## 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/2:filtLen/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;

disp(['Varians af input signal x: ' num2str(var(xM))]);

disp(['Fejl-varians af det lavpas filtrede 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('Lavpas filtrede 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);

# Appendix B – VHDL og SOPC overblik

## VHDL kode oversigt

Dette appendix indeholder en oversigt over VHDL koden for komponenterne og tilhørende test bench inkluderet i **VHDLCode.zip** filern. Der 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
* **audioIIR\_st**.vhd: IIR filter som kun er testet I ModelSim (Ikke beskrevet i rapporten)
* **audioIIR\_st\_tb**: Test bench

**Komponenter til ST Bus Version 3:**

(Rune – opdateres komponenter mangler)

* **audio\_process\_st3.vhd**: Audio Process til ST bus:
* **audio\_process\_st3\_tb.vhd**: Test bench - mangler
* **multiplexer.vhd**: ST Bus multiplexer for stereo - mangler
* **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

## 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)

# References

1. Woon-seng gan, Emmbedded 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](http://www.altera.com)
6. Avalon Interface Specification: <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf>