AARHUS
UNIVERSITET
INGENIØRHØJSKOLEN

Komponenter til lydbehandling i en FPGA

ETDSPC Projekt

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

Q1 2012

Indhold

1 Indledning 3
1.1 Report struktur 3
2 Problemformulering 4
3 Udviklingsstrategi 5
4 Arkitektur 7
5 Komponent beskrivelser og validering 8
5.1 ModelSim Testbench 8
5.2 FIR filter 12
5.3 Sigma delta converter 12
5.4 LMS filter 12
5.5 Delay 19
6 SoPC systemerne 23
6.1 Sigma delta SoPC 23
6.2 LMS filter SoPC 23
7 Test resultater 26
7.1 Diskussion af resutater 27
7.2 Forslag til forbedringer 27
8 Konklusion 27
9 Appendix 28
10 References 28

1 Indledning

(Rune)

- Hvorfor denne rapport – læringsmål
- Hvilke læringsmål er dækket af dette projekt

Opsætning af mål:

- Implementere programmer for FPGA'er, skrevet i VHDL
  - OK
- Anvende modelsim og test benches til at udføre simulation af VHDL design
  - Beskrive testbench for LMS og SigmaDelta converter
  - Assertion – hvordan kan vi få det med ?
- Anvende constraints til specifikation af system krav
  - Introduktion til problemet, eg. 48 khz med 12 Mhz – latency, throughput, area
- Redegøre for begreber som: clock domæner, clock skew, pipelining, PLL- og memory komponenter
- 48 khz vs. 1.2 mhz sigmadelta

- Redegøre for timings simulering og analyse i Quartus II værktøjet
  - Ikke anvendt
- Anvende soft cores til opbygning af et SoC (System On Chip) system
  - OK
- Implementere C programmer til afvikling på SoC
  - OK
- Implementere signal behandlings algoritmer i VHDL
  - Absolut, filter mm.

1.1 Report struktur

2 Problemformulering

(Kim)

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 Alteras Streaming Bus (ST). Målet med dette projekt er at implementere forskellige audio lydbehandlings komponenter, der kan benyttes i et Altera SOPC design. Komponenters opsætning skal kunne konfigureres med brug af VHDL generics. Ændring af parameter skal kunne styres fra Nios II processoren. Hertil benyttes Alteras Memory Mapped Bus (MM). 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 interface til processoren (MicroBlaze). Dette krævet at vi ikke benytter det indbygget Altera komponent biblioteker men implementere vores egne filter, audio komponenter og 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
        - Her er målet 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 delay
  - Sigma Delta i stereo med aktivt analog filter
- Oversigt af udviklede audio komponenter (ST, MM kompatible) med information om ressource forbrug som area (LE, Multipliers, Block RAM), latency, throughput
- Denne rapport med beskrivelse af arkitektur, implementering samt refleksioner over resultater og læring

3 Udviklingsstrategi

(Kim)

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 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 den maksimum clock frekvens (Fmax).

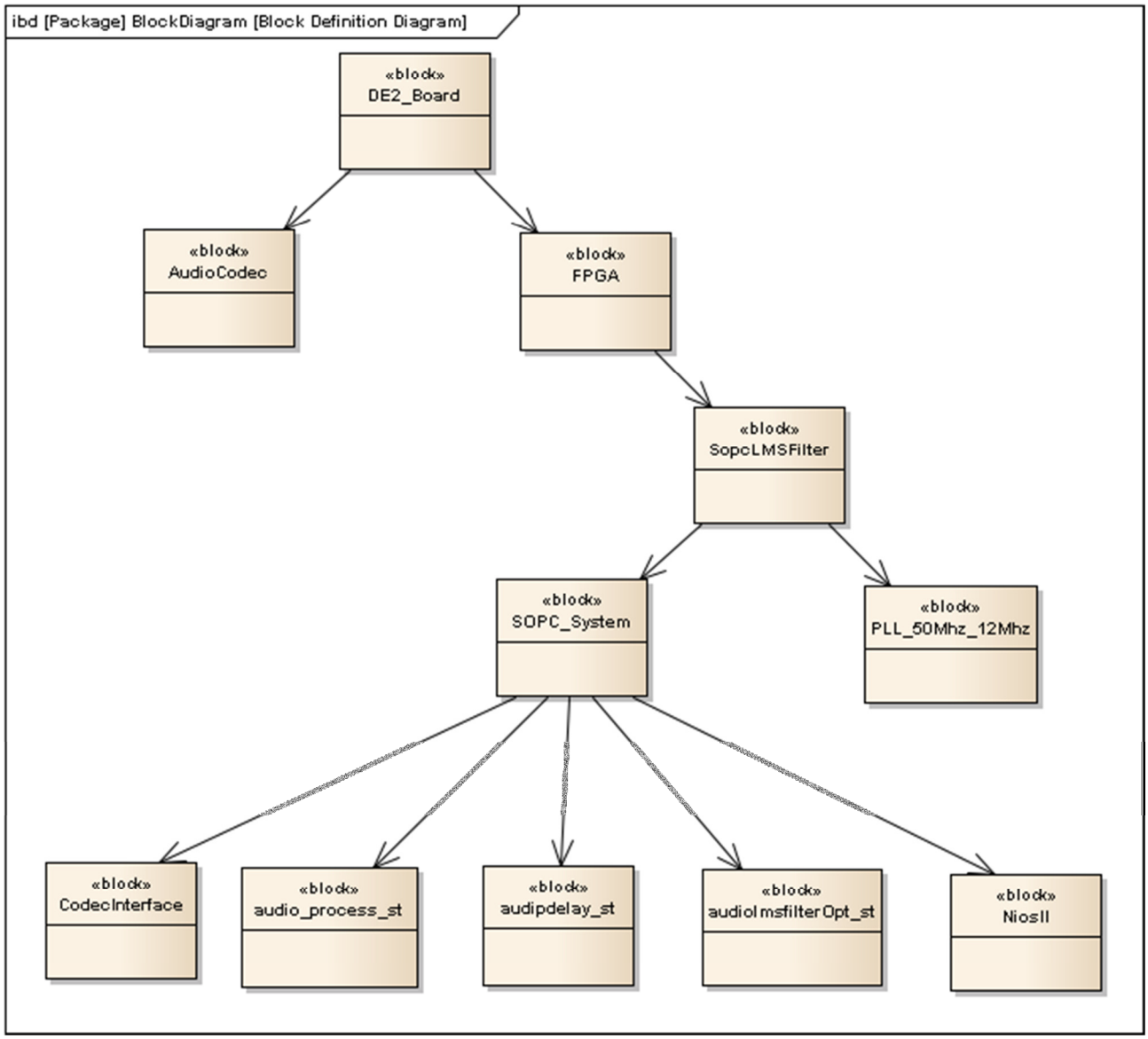
Der er udviklet flere forskellige SoPC projekter med Nios II processoren, hvor vi løbende har indsat versioner af komponenterne efter afprøvning i ModelSim. Det har vist sig at være en god strategi. Selv om komponenten er testet i ModelSim er det ikke altid det virkede på DE2 boardet. Det kunne f.eks. 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 ikke virkede.

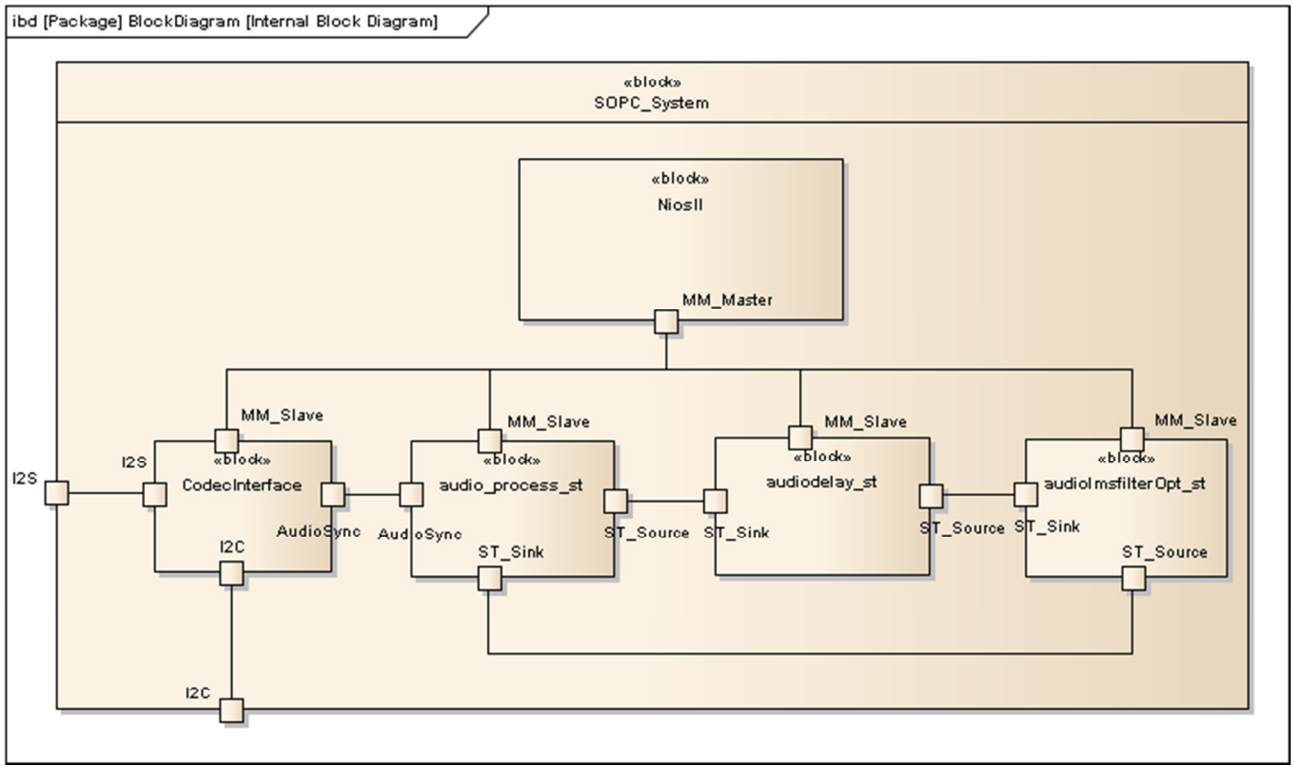
Vi startede med et SoPC design, hvor vi har benyttet et VHDL komponent udleveret i kurset, med konvertering fra I2S til ST format. Et separat I2C komponent foretager kommunikation og initialisering af codec via. softwaren på Nios II processoren. Dette første SoPC projekt har vi haft store vanskeligheder 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 stereo sigma delta konverter og LMS filteret med stereo delay.

- Fra formel til Matlab og/eller C-reference
- Stepvis udvikling af SoPC versioner
- Stepvis udvidelse af ModelSim versioner
- Håndtering af fejl i forhold til simulering vs. SoPC på target
- Version før ST interface introduktion
- Læsevejledning til efterfølgende afsnit

4 Arkitektur

(Rune)





- Audio, ST og MM Bus
- Block diagrammer (SysML) - Kim
- Eksempel på vores implementation ST og MM bus
- Audio streaming – Codec interface (I2S -> Sync Bus og I2S -> ST Bus)
- Opsætning af Code på 2 forskellige metoder (HW vs. SW)
- ST bus vs. simplet 48 kHz sync interface -> flere komponenter

5 Komponent beskrivelser og validering

(Kim)

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 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;
```

5.2 FIR filter

(Rune)

- Forskellige typer (Direct form 1, Transponsed, Sysmetrisk)
- Optimering for area

5.3 Sigma delta converter

(Rune)

5.4 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.

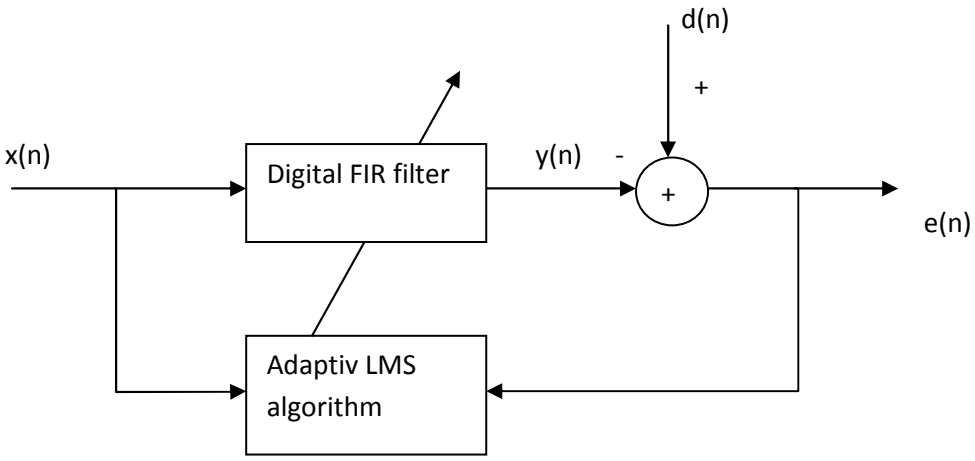


Figure 1 Adaptiv LMS filter

Det digitale FIR filter beregnes som

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

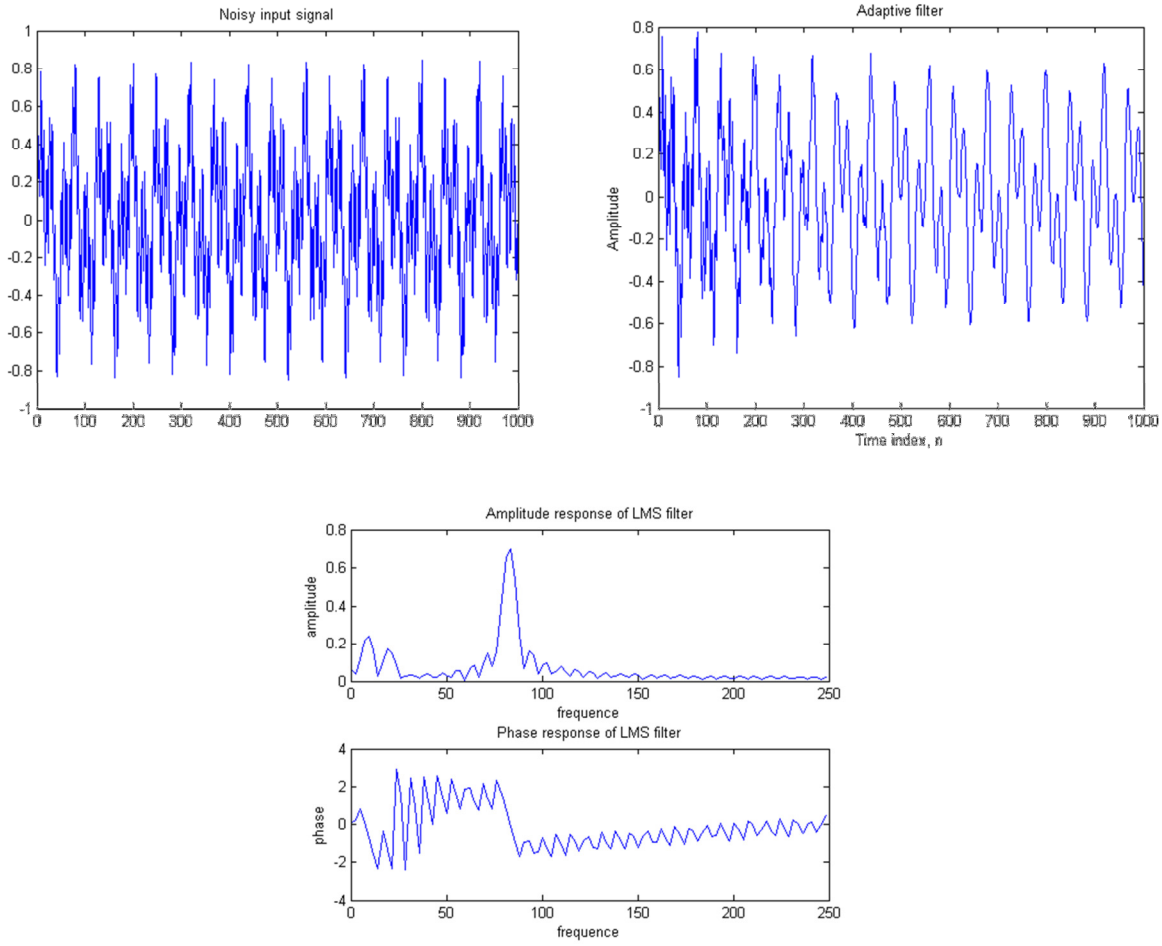
LMS algoritmen beregner nye koefficienter efter formlen, hvor μ er en meget lille adaptions konstant

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

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 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 1 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
    }
```

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.

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;
```

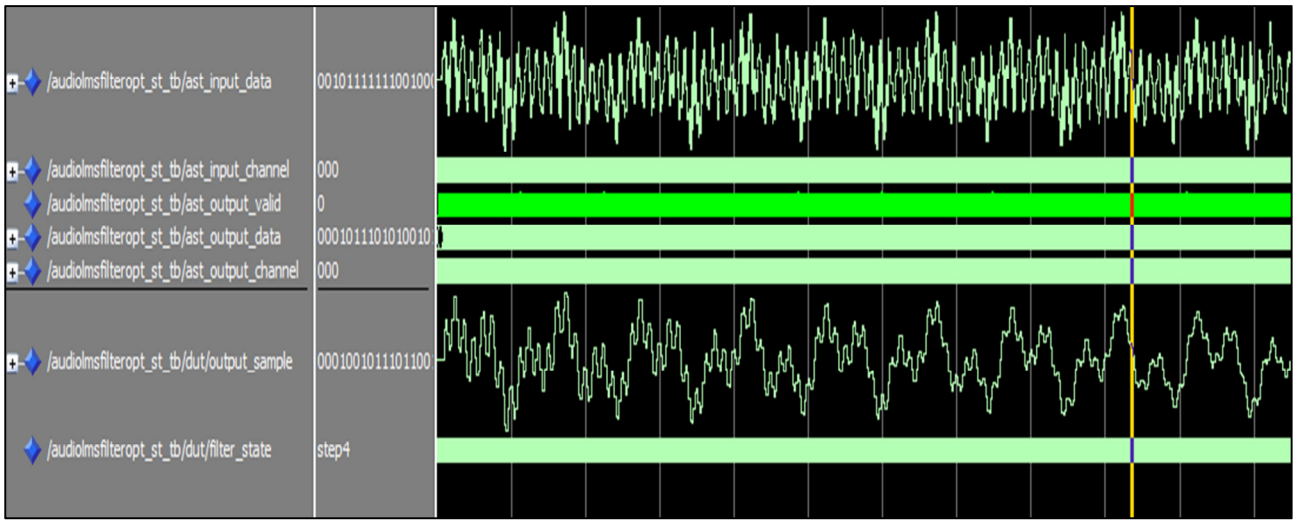
Først skiftes "Tapped Delay Line" for FIR filteret, der er implementeret i "Direct Form". Det er ikke muligt at benytte hverken en "Transposed Structure" eller en symmetrisk implementering. Den "Tapped Delay Line" skal efterfølgende bruges i udregningen af de nye koefficienter. Det pipelined filter består 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
```

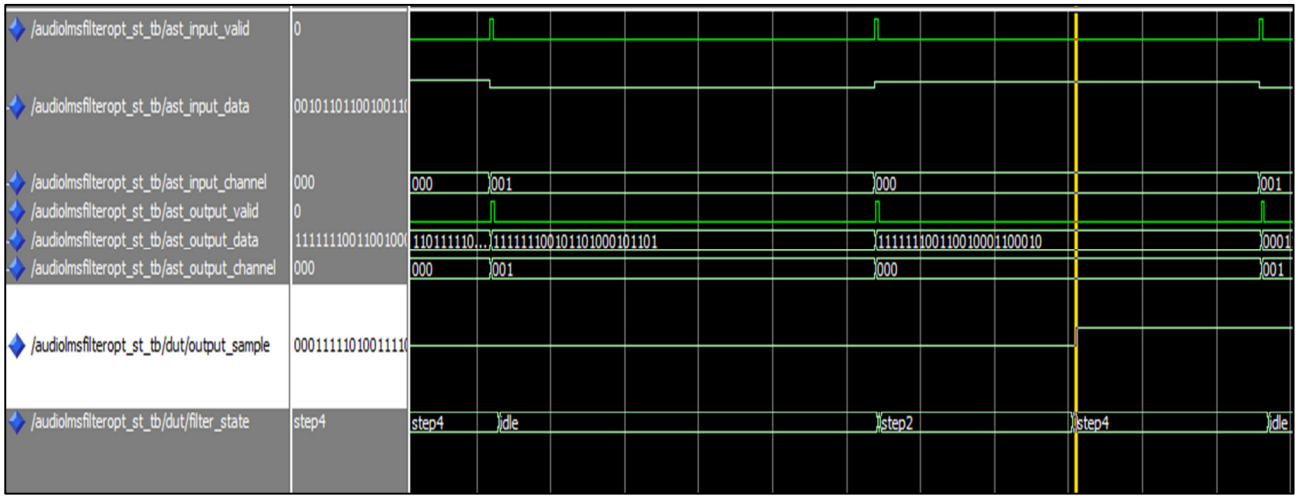
```
      case filter_state is

        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;
```

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 3 viser resultatet af timings simuleringer i ModelSim, hvor det kan ses på **output_sample** signalet at LMS filteret justeret sig på plads.



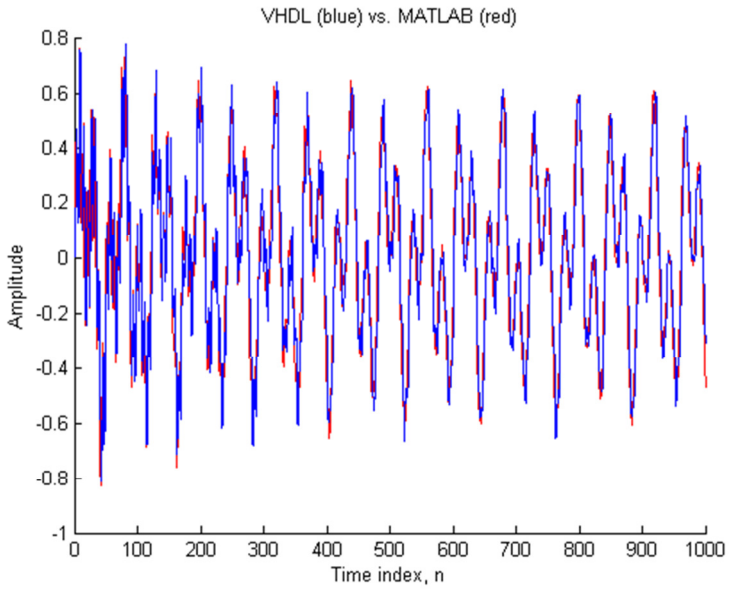
Figur 2 Verifikation af optimeret LMS filter med ST bus interface

Figur 4 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 3 LMS filter med ST bus interface – audio samples for højre og venstre kanal

Figur 4 viser outputtet fra det simulerede LMS filter med 24 bits fixed point sammenlignet med versionen i double præcision fra MATLAB:



Figur 4 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	166 ns	speed	70	11354 (34 %)	876	28.66 MHz
LMS area audiolmsfilterOpt_st .vhd (ST bus)	64	11 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.

5.5 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;
```

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;
```

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

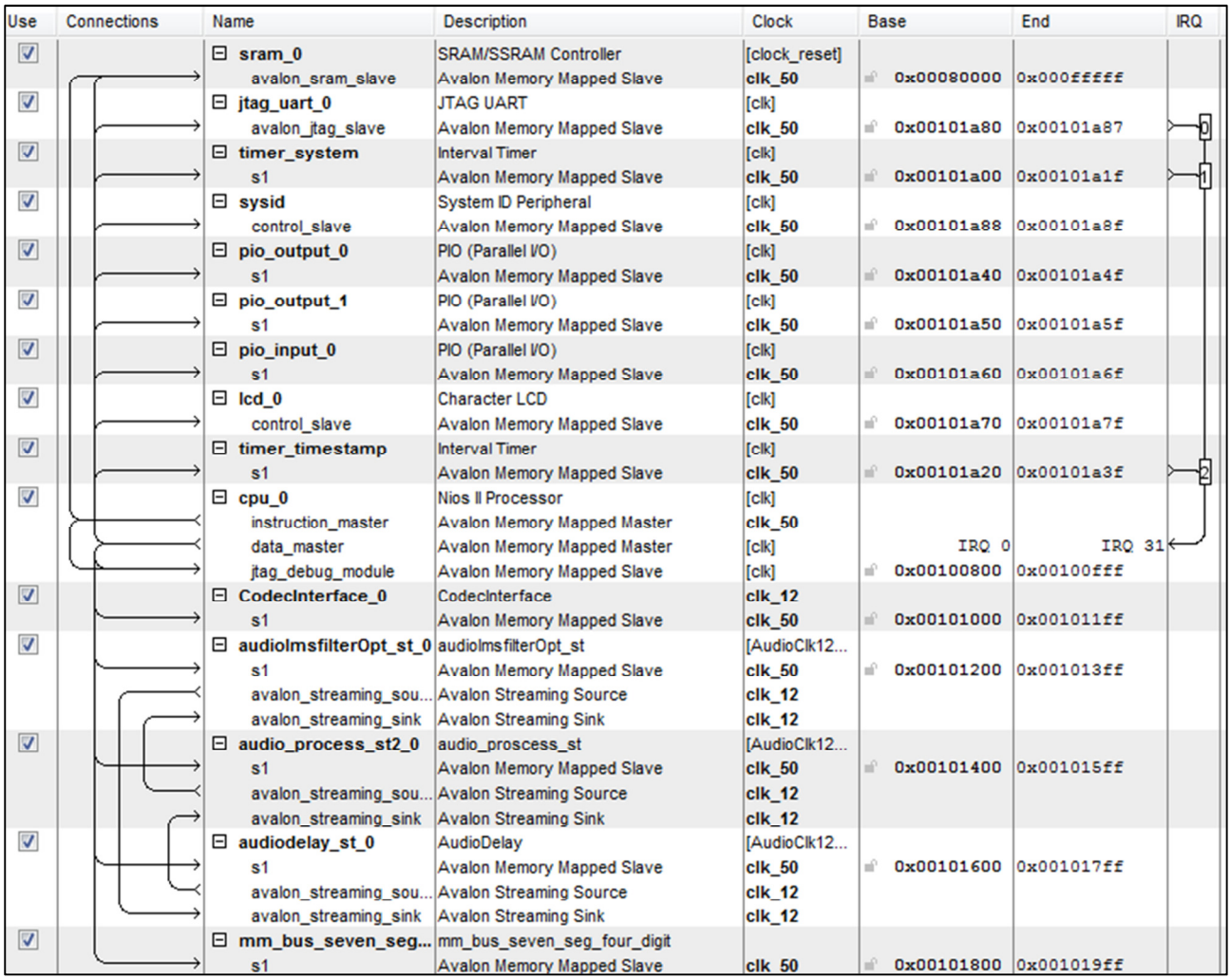
6 SoPC systemerne

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.

6.1 Sigma delta SoPC (Rune)

6.2 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.



Use	Connections	Name	Description	Clock	Base	End	IRQ
☑		sram_0	SRAM/SSRAM Controller	[clock_reset]			
		avalon_sram_slave	Avalon Memory Mapped Slave	clk_50	0x00080000	0x000fffff	
☑		jtag_uart_0	JTAG UART	[clk]			
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_50	0x00101a80	0x00101a87	0
☑		timer_system	Interval Timer	[clk]			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101a00	0x00101a1f	1
☑		sysid	System ID Peripheral	[clk]			
		control_slave	Avalon Memory Mapped Slave	clk_50	0x00101a88	0x00101a8f	
☑		pio_output_0	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101a40	0x00101a4f	
☑		pio_output_1	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101a50	0x00101a5f	
☑		pio_input_0	PIO (Parallel I/O)	[clk]			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101a60	0x00101a6f	
☑		lcd_0	Character LCD	[clk]			
		control_slave	Avalon Memory Mapped Slave	clk_50	0x00101a70	0x00101a7f	
☑		timer_timestamp	Interval Timer	[clk]			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101a20	0x00101a3f	2
☑		cpu_0	Nios II Processor	[clk]			
		instruction_master	Avalon Memory Mapped Master	clk_50			
		data_master	Avalon Memory Mapped Master	[clk]		IRQ 0	IRQ 31
		jtag_debug_module	Avalon Memory Mapped Slave	[clk]	0x00100800	0x00100fff	
☑		CodecInterface_0	CodecInterface	clk_12			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101000	0x001011ff	
☑		audioImsfilterOpt_st_0	audioImsfilterOpt_st	[AudioClk12...			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101200	0x001013ff	
		avalon_streaming_sou...	Avalon Streaming Source	clk_12			
		avalon_streaming_sink	Avalon Streaming Sink	clk_12			
☑		audio_process_st2_0	audio_proscess_st	[AudioClk12...			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101400	0x001015ff	
		avalon_streaming_sou...	Avalon Streaming Source	clk_12			
		avalon_streaming_sink	Avalon Streaming Sink	clk_12			
☑		audiodelay_st_0	AudioDelay	[AudioClk12...			
		s1	Avalon Memory Mapped Slave	clk_50	0x00101600	0x001017ff	
		avalon_streaming_sou...	Avalon Streaming Source	clk_12			
		avalon_streaming_sink	Avalon Streaming Sink	clk_12			
☑		mm_bus_seven_seg...	mm_bus_seven_seg_four_digit				
		s1	Avalon Memory Mapped Slave	clk_50	0x00101800	0x001019ff	

Figur 5 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 6 Forbindelser af SOPC_Systemet med DE2 boardet

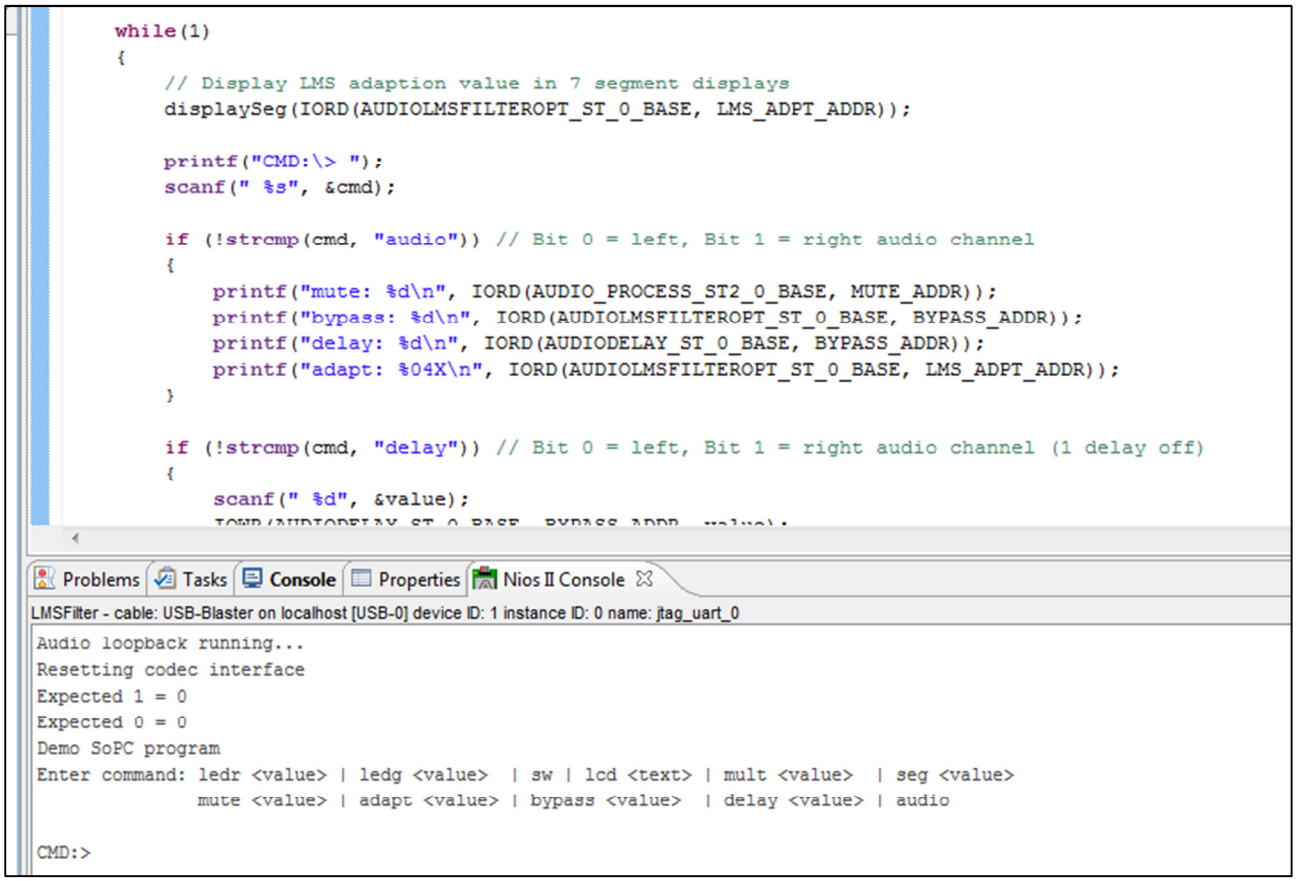
Softwaren, der er skrevet til Nios II processoren, implementere en simpel menu, hvor det er muligt at give kommandoer til ændring af hardware komponenternes funktioner i real-time.

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 7 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 8 FPGA resource forbrug

7 Test resultater

- Dump af ModelSim simuleringerne
- Reference til Video for HW test

7.1 Diskussion af resutater

7.2 Forslag til forbedringer

8 Konklusion

9 Appendix A

10 Appendix B

- Oversigt over modeler, kode, arkiver mm.
- ModelSim tests:

Komponeter til Sync Bus:

- Grundlaget er CodecInterface
- Audio Process til Sync bus: audio_process.vhd
- Symetric FIR Filter med Sync bus: audiofilter.vhd
- Transposed FIR Filter med Sync bus: audiotransposedfilter.vhd

Kompontere til ST Bus V2:

- Audio Process til ST bus: audio_process_st2.vhd
- LMS filter med Sync bus: audiolmsfilter.vhd
- LMS filter optimeret med ST bus: audiolmsfilterOpt_st.vhd
- Audio delay med ST bus: audiodelay_st.vhd, delay_ram.vhd

Kompontere til ST Bus V3:

- Audio Process til ST bus: audio_process_st3.vhd
- ST Bus multiplexer: multiplexer.vhd
- SigmaDelta Converter: UpSampler.vhd

Test bench utility:

- Txt_util.vhd

MATLAB og Simulink utilities:

11 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]