NTNU
Det skapende universitet

# Realisering av high-end delta-sigma DAC i FPGA

**Lasse Haugnes Olsen**

Master i elektronikk
Oppgaven levert: Juli 2006
Hovedveileder: Trond Sæther, IET
Biveileder(e): Ivar Løkken, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

# Oppgavetekst

En vesentlig fordel med delta-sigma DACer er muligheten for å bruke et to-nivå utsignal som tillater
en full digital implementasjon av konverteren. Dette er meget fordelaktig med tanke på realisering i
FPGA eller nanoskala digitalprosesser for laveffektapplikasjoner. Imidlertid har denne type
realiseringer tradisjonelt vært hemmet av jitterfølsomhet og intersymbolinterferens. I oppgaven vil
utfordringer i forbindelse med, og fordelaktige implementasjoner av, en fulldigital delta-sigma DAC
for audiobruk studeres. Videre vil en fullstendig DAC implementeres i et Xilinx FPGA-system. Målet
vil være å lage en fungerende DAC med høy hifi-ytelse.

Oppgaven gitt: 30. januar 2006
Hovedveileder: Trond Sæther, IET

# Sammendrag

Oppgaven har gått ut på å lage en *high-end* delta-sigma DA-konverter for FPGA.

Prinsippet for en slik DA-konverter er at det analoge utgangssignalet genereres ved å lavpassfiltrere et 1-bit digitalt utgangssignal fra FPGA-kretsen. For å oppnå ønsket signalkvalitet ved den 1-bit representasjonen, krever dette at det benyttes oversampling og støyforming ved hjelp av en delta-sigma-modulator.

I det 1-bit utgangssignalet er benyttet pulsbredde-modulasjon (PWM), som er gunstig med tanke på de ikke-ideelle egenskaper på utgangen av FPGA-kretsen og i det analoge lavpassfilteret.

På bakgrunn av oppgavebeskrivelsen ble det satt som mål at DA-konvertern skal kunne oppnå THD+N bedre enn -100 dB, samt kunne benyttes med en samplerate på 44.1 kHz som tilsvarer CD-lyd.

DA-konverteren er realisert i verilog. Simuleringer viser at denne vil kunne oppnå en THD+N på -98 dB, som nok kan anses som *high end,* mens DA-konverteren bare vil kunne benyttes med en samplefrekvens på rundt 20 kHz, som ikke kvalifiserer til denne betegnelsen.

Målene i oppgavebeskrivelsen er altså bare delvis oppfylt.

# Forord

Denne rapporten er et resultat av masteroppgave utført ved linjen for Krets og systemkonstruksjon – institutt for elektronikk og telekommunikasjon, NTNU, våren 2006.

Jeg vil rette en stor takk til veileder Ivar Løkken som alltid har vært positiv og hjelpsom, samt til mine medstudenter på fokus lesesal for mang en hyggelig burger-pause i ukene før innleveringen av oppgaven.

Trondheim 3. juli, 2006

Lasse Olsen

# Innholdsfortegnelse

# Figurliste

# Tabeller

# Forkortelser

| | |
|---|---|
| **BIBO** | *Bounded Input – Bounded Output* |
| **CD** | *Compact Disc* |
| **CIC** | *Cascade Integrator Comb* |
| **CIFB** | *Cascade of Integrators, Feedback form* |
| **DA** | *Digital-Analog* |
| **DS** | *Delta-sigma* |
| **DVD** | *Digital Versatile Disc* eller *Digital Video Disc* |
| **FIR** | *Finite Impulse Response* |
| **FPGA** | *Field Programmable Array* |
| **IIR** | *Infinite Impulse Response* |
| **INS** | *Integral Noise Shaping* |
| **MAC** | *Multiply Accumulate* |
| **$N_{TF}$** | *Noise Transfer Function* |
| **OSR** | *Ove Sampling Ratio* |
| **PCM** | *Pulse Code Modulation* |
| **PD** | *Predistortion* |
| **PDM** | *Pulse Density Modulation* |
| **PWM** | *Pulse Width Modulation* |
| **RAM** | *Random Access Memory* |
| **ROM** | *Read Only Memory* |
| **SNR** | *Signal to Noise Ratio* |
| **SP-DIF** | *Sony Philips Digital Interface* |
| **$S_{TF}$** | *Signal Transfer Function* |
| **THD** | *Total Harmonic Distortion* |
| **THD+N** | *Total Harmonic Distortion + Noise* |

# Begrepsforklaringer

| | |
|---|---|
| Sample | Punktprøve av kontinuerlig signal |
| Nyquist-rate | Samplerate, samplefrekvens, punktprøvingsfrekvens |
| Nyquist-bånd | Frekvensbånd fra 0 til 0.5 · Nyquist-rate |
| Basisbånd | Område av Nyquist-bånd hvor signal av interesse ligger |
| $F_s$ | Fysisk samplefrekvens |
| $f_s$ | Normalisert samplefrekvens |
| F | Fysisk frekvens |
| f | Normalisert frekvens |
| $f_0$ | Normalisert øvre basisbånd-frekvens |

# 1 Innledning

## 1.1 Oppgavebeskrivelse og avgrensning

Oppgavebeskrivelsen lyder som følger:

*"Implementering av high-end delta-sigma audio DAC i FPGA"*

Utover denne tittelen er det ikke angitt spesifikke krav den ferdige DA-konverteren skal overholde. Det har likevel vært nødvendig å nedfelle en mer konkretisert liste med spesifikasjoner som utgangspunkt for det videre arbeidet med oppgaven.

*Så hva er High-End?*

For å svare på dette er det naturlig å ta utgangspunkt i de begrensningene som allerede ligger i dagens formater for digital lyd samt spesifikasjonene for eksisterende produkter tilsvarende det som skal lages i denne oppgaven.

Når det gjelder formatet på digital lyd er det vanlig med en oppløsning på 16 bit for CD [33] eller opp mot 24 bit for DVD [35], med samplefrekvenser på henholdsvis 44.1 kHz og 96 kHz. Ved digitalisering av lyd vil det alltid oppstå feil som følge av at det analoge signalet kvantiseres. Dette avviket fra det analoge signalet vil framstå som støy i det digitale signalet. Nivået på denne støyen vil være gitt av antallet nivåer lyden tilnæres til, det vil si antall bit lyden er representert med etter digitaliseringen. For 16 bit CD-lyd kan man anta at kvantiseringsstøy vil begrense signal-støyforhold (SNR) til rundt 98 dB, mens det tilsvarende for 24-bit DVD-lyd vil være rundt 146 dB.

I tillegg til den rene kvantiseringsstøyen, vil det av ulinearitet i enkelte komponenter og konverteringsprosesser kunne oppstå harmonisk forvrengning (THD), det vil si harmoniske frekvenskomponenter av inngangssignalet.

I [16] foretas det en sammenlikning av 6 state-of-the-art DA-konvertere. SNR for disse ligger i området 117 til 127 dB, mens THD+N ligger i området -100 til -108 dB.

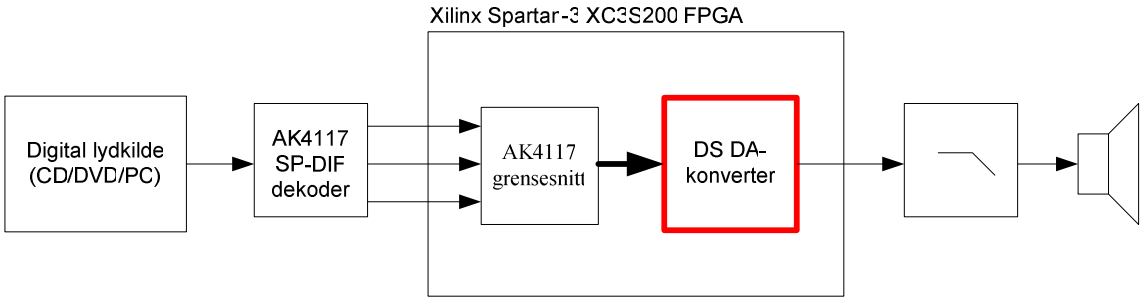
Ettersom kretsen skal realiseres i FPGA impliserer dette at DA-omformingen gjøres ved hjelp av lavpassfiltrering av et 1-bit digitalt utgangssignal fra denne. Konstruksjon av dette eksterne lavpassfilteret er ikke ment som en del av oppgaven.

På bakgrunn av punktene ovenfor har målspesifikasjonene for DA-konverteren blitt satt til:

- SNR > 120 dB
- THD+N < -100 dB
- Må kunne brukes med samplefrekvens $\geq$ 44.1 kHz
- 1-bits utgangssignal direkte kopling av FPGA til eksternt lavpassfilter

## 1.2 Utviklingsoppsett

Figur 1 viser det planlagte utviklingsoppsettet for prosjektet, bestående av en AK4117 SP-DIF dekoder for innhenting av data fra digital lydkilde, en Spartan-3 XC3S200 FPGA for implementering samt et lavpassfilter for filtrering av 1-bit digitalt utgangssignal. Det vil i prosjektet bare bli lagt vekt på konstruksjon av kjernen for delta-sigma DA-konverteren (i rød ramme), og derfor benyttes ferdig hardware/software for de resterende modulene i oppsettet.



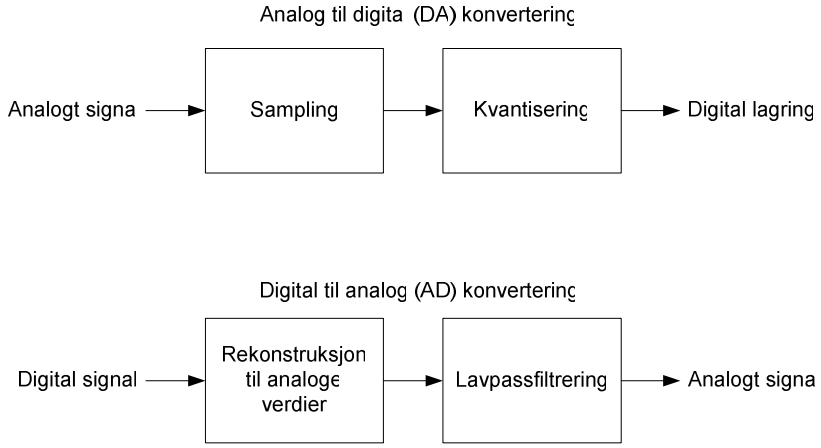

**Figur 1. Utviklingsoppsett**

# 2 Bakgrunn

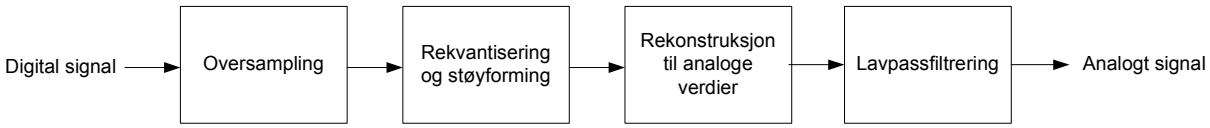
*Det vil i dette kapittelet bli gitt en innføring i teori som er nødvendig for å få fullt utbytte av denne rapporten Et sentralt tema for rapporten er DA/AD-konvertering, og spesielt DA-konvertering med rekvantisering og støyforming i en delta-sigma (DS) modulator.*

Figur 2 *viser prinsippet for AD/DA-konvertering. Ved AD-konvertering tas det sampler, eller punktprøver, av det analoge signalet, som så blir kvantisert for digital representasjon. Ved en AD-omformer rekonstrueres de kvantiserte punktprøvene som analoge verdier, som deretter blir lavpassfiltrert for å gjenskape det analoge signalet.*

Figur 3 *viser prinsippet for en delta-sigma DA-konverter. Her oversamples og rekvantiseres det digitale signalet til et lavere antall diskrete nivåer før rekonstruksjon av de analoge nivåene. Kombinasjonen av oversampling og støyforming i en DS-modulator gjør dette mulig uten at signalkvaliteten behøver og forringes nevneverdig. En vesentlig fordel med denne typen DA-konvertering er at dette reduserer antallet analoge nivåer som må rekonstrueres.*




**Figur 2. Prinsipp for AD/DA-konvertering.**




**Figur 3. Prinsipp for DA-konvertering med støyforming i delta-sigma modulator.**

## 2.1 Definisjon SNR, THD , SNRD og THD+N

### 2.1.1 SNR

*Signal to Noise Ratio* (SNR), eller signal-støy forhold, er definert som forholdet mellom maksimal signaleffekt og total støyeffekt i basisbåndet, og uttrykkes som:

$$SNR = \frac{P_{signal}}{P_{støy}} = \left( \frac{A_{signal}}{A_{støy}} \right)^2 \quad \textbf{(1)}$$

eller i dB som

$$SNR = 10\,log\left( \frac{P_{signal}}{P_{støy}} \right) = 20\,log\left( \frac{A_{signal}}{A_{støy}} \right) \quad \textbf{(2)}$$

der P er støyeffekt og A er RMS signalamplitude (gjelder både for signal og støy).

Det er ofte ønskelig med høy SNR da dette innebærer minst støy.

### 2.1.2 THD

*Total Harmonic Distortion* (THD) er definert som total effekt til alle harmoniske komponenter i forhold til maksimal signaleffekt:

$$THD = \frac{\sum P_{harmoniske}}{P_{signal}} = \frac{\sum A_{harmoniske}{}^2}{A_{signal}{}^2} \quad \textbf{(3)}$$

eller i dB som

$$THD = 10\,log\left( \frac{\sum P_{harmoniske}}{P_{signal}} \right) = 10\,log\left( \frac{\sum A_{harmoniske}{}^2}{A_{signal}{}^2} \right) \quad \textbf{(4)}$$

hvor P er støyeffekt og A er RMS signalamplitude (gjelder både for signal og støy).

### 2.1.3 THD+N

*Total Harmonic Distortion + Noise* (THD+N) angir forholdet mellom all støy, det vil si både overharmonisk og generell bakgrunnsstøy, og signaleffekt. THD+N uttrykkes som:

$$THD + N = \frac{\left(\sum \mathrm{P}_{harmoniske}\right) + P_{støy}}{P_{signal}} = \frac{\left(\sum \mathrm{A}_{harmoniske}{}^{2}\right) + A_{støy}{}^{2}}{A_{signal}{}^{2}} \quad \textbf{(5)}$$

eller i dB som

$$THD + N = 10\log\left(\frac{\left(\sum \mathrm{P}_{harmoniske}\right) + P_{støy}}{P_{signal}}\right) = 10\log\left(\frac{\left(\sum \mathrm{A}_{harmoniske}{}^{2}\right) + A_{støy}2}{A_{signal}{}^{2}}\right) \quad \textbf{(6)}$$

Det er ofte ønskelig med lav THD/THD+N da dette innebærer minst støy.

### 2.1.4 Generelt

Vi ser at SNR og THD+N er nært tilknyttet. THD+N angir også et signal-støy forhold, og kan sammenliknes med 1/SNR (som tilsvarer –SNR i dB) direkte.

Selv om THD+N angir signal-støy-forhold der alle støykomponentene er medregnet, blir det likevel ved audio ofte oppgitt spesifikasjoner både for SNR og THD+N. Siden overharmonisk støy er et produkt av signalet i seg selv må en fullstendig spesifikasjon av THD+N også inneholde informasjon om dette. Ved spesifisering av THD/THD+N benyttes ofte følgene begreper:

- dBFs, som angir fullskala referansenivå
- dBr, som angir signalnivået relativt til referansenivå

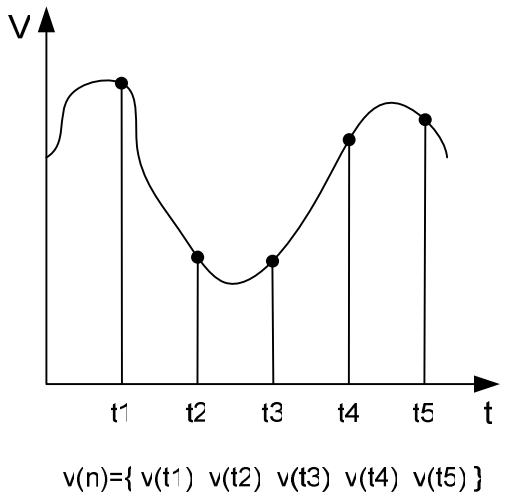
Båndbredden spesifikasjonene gjelder for bør også spesifiseres. Ofte oppgis imidlertid ikke denne der det er implisitt at spesifikasjonene gjelder for hele systembåndbredden.

Hva som er basisbåndet defineres ofte etter et såkalt A-veiingsfilter (A-weighted) som er gitt av standard [34]. Alternativt er ofte basisbåndet definert som 0-20kHz, som er det antatte frekvensområdet for menneskets hørsel.

## *2.2 Sampling og kvantisering*

### 2.2.1 Prinsipp

Et analogt signal kan representeres digitalt ved hjelp av et sett punktprøver, eller samples, av det analoge signalet. Figur 4 illustrerer et analogt signal v(t) som blir representert som v(n) ved hjelp av 5 samples.




**Figur 4. Diskret representasjon av kontinuerlig signal.**

Det er åpenbart at man ved digitalisering vil miste noe av informasjonen fra det analoge signalet da det alltid vil være begrenset hvor ofte signalet kan samples og hvor nøyaktig hver enkelt sample kan angi signalnivået i et punkt. Denne tilnærmingen av det kontinuerlige signalet til et diskret antall nivåer i tid og amplitude kalles kvantisering.

Unøyaktigheten, eller avviket som følge av at en sample ikke kan gjengi signalnivået i et punkt med uendelig nøyaktighet, kan sees på som støy i det digitale signalet. Det nødvendige antall nivåer, eller antall bit, som må brukes for å representere hver sample, vil være gitt av hvor mye slikt kvantiseringsstøy som kan tolereres.

Representasjon av et analogt signal ved hjelp av samples tatt med like (uniforme) intervaller omtales ofte som *Pulse Code Modulation* (PCM).

### 2.2.2 Samplerate

*Nyquist-Shannons sample-teorem* sier at et signal må være båndbegrenset og samples med en frekvens lik minimum det dobbelte av signalbåndbredden for å kunne rekonstrueres feilfritt [25].

Sampleraten, eller samplefrekvensen, for et signal omtales ofte som Nyquist-raten. Det kan altså samples og gjengis signaler med frekvenser som er lavere en halve Nyquist-raten.

Frekvensbåndet som kan gjengis omtales ofte som Nyquist-båndet.

### 2.2.3 Normalisering av frekvensrepresentasjon

Ved digital signalbehandling blir som oftest den fysiske frekvensen F normalisert i forhold til Nyquist-raten $F_s$ på en av følgende måter:

1. $$f = \frac{F}{F_s}$$ som gir Nyquist-bånd f = 0 → 0.5

2. $$\omega = \frac{2 \cdot \pi \cdot F}{F_s}$$ som gir Nyquist-bånd ω = 0 → π

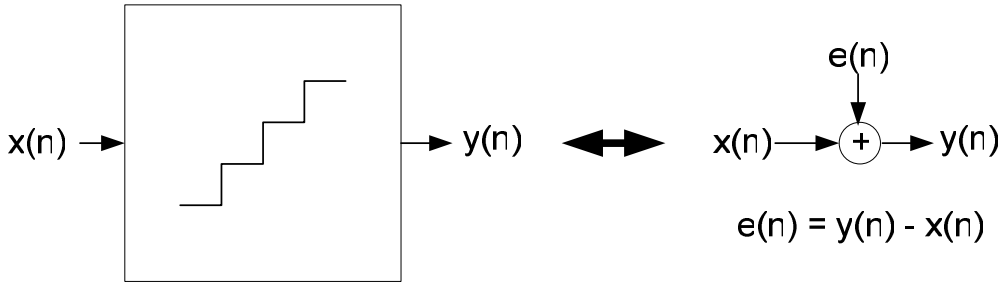
3. $$f = \frac{2 \cdot F}{F_s}$$ som gir Nyquist-bånd f = 0 → 1

Representasjonene 1 og 2 er nok de mest vanlige brukt i litteratur om digital signalbehandling, mens representasjon 3 er vanlig i Matlab.

I denne rapporten benyttes representasjon 1 eller 2 hvis ikke annet er spesifisert.

### 2.2.4 Modell for analyse av kvantiseringsstøy

For å finne det nødvendige antallet kvantiseringsnivåer, eller antall bit, er det vanlig å sette opp en modell for å analysere kvantiseringsstøyen. Figur 5 viser Bennets støymodell [26] som er den mest utbredte modellen for kvantiseringsstøy. Her sees avviket ved samplingen på som en støykomponent e(n) som tilføres inngangssignalet x(n).

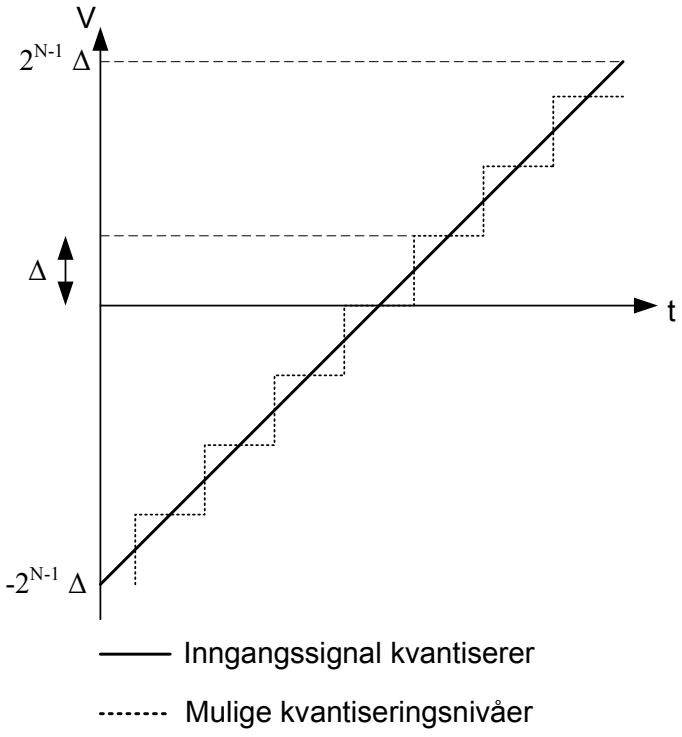



**Figur 5. Bennets støymodell for analyse av kvantiseringsstøy.**

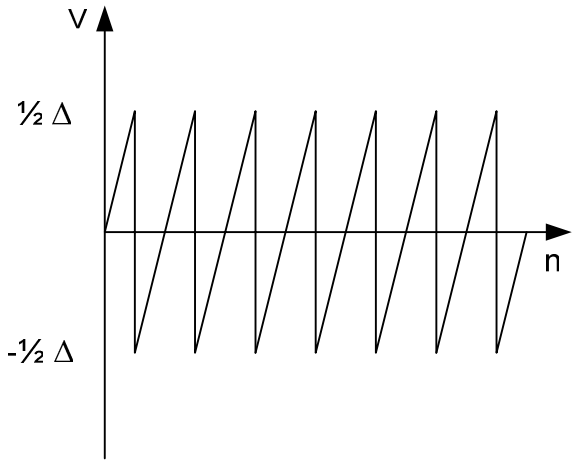
### 2.2.5 Beregning av effekt i kvantiseringsstøy

I modellen for kvantiseringsstøy er det naturlig å anta at avviket e(n) mellom det korrekte analoge nivået y(n) og sampleverdien x(n) vil måtte ligge mellom -½ Δ og ½ Δ, der Δ angir den minste avstanden mellom hvert av de mulige nivåene sampleverdien kan innta. Figur 7 illustrerer N=3 bits kvantisering av et rampeformet signal. Dette vil gi en kvantiseringsstøy e(n) som vist i Figur 7.

Kvantiseringsstøyen i Figur 7 er funnet med et deterministisk inngangssignal, men det er også vanlig å anta denne formen på e(n) ved en stokastisk tilnærming. Dette forutsetter et inngangssignal med en variasjon som gjør det rimelig å anta at e(n) vil innta tilfeldige verdier uniformt fordelt mellom ±1/2 Δ [26].




**Figur 6. N=3 bits kvantisering av rampeformet signal.**




**Figur 7. Form på kvantiseringsstøy e(n).**

Det antas altså at kvantiseringsstøyen vil innta form som en sagtannpuls med amplitude lik ½ Δ. Med utgangspunkt i dette er det rett fram å finne følgende uttrykk for kvantiseringsstøyeffekten $P_q$:

$$P_q = \frac{1}{\Delta}\int_{-\frac{\Delta}{2}}^{\frac{\Delta}{2}} V^2 \, \mathrm{d}t = \frac{1}{12}\Delta^2 \qquad (7)$$

## 2.2.6 Kvantiseringsstøy og SNR

Vi ser i uttrykket for kvantiseringsstøy i (7) at denne vil være uavhengig av inngangssignalet, og bare gitt av avstanden mellom hvert kvantiseringsnivå Δ. Med bakgrunn i dette kan vi finne sammenhengen mellom antall bit kvantisering N og det maksimale signal-støy-forholdet (SNR) etter kvantisering.

Siden kvantiseringsstøyen er uavhengig av signalnivået vil man få best SNR ved maksimal amplitude på inngangssignalet. I tillegg vil beregnet SNR være avhengig av hvilken form man antar inngangssignalet har. Hvis vi antar at inngangssignalet har sagtannform med maksimal amplitude lik $\Delta\, 2^{(N-1)}$ får vi følgende uttrykk for maksimal SNR:

$$SNR_{maks} = 10\log\left(\frac{P_{signal}}{P_{støy}}\right) = 10\log\left(\frac{\frac{\Delta^2 2^{2N}}{12}}{\frac{\Delta^2}{12}}\right) = 10\ \log\left(2^{2N}\right)$$

$$= 6.02N \tag{8}$$

Tilsvarende uttrykk ved sinusformet inngangssignal vil være:

$$SNR_{maks} = 10\log\left(\frac{\left[\frac{\Delta\, 2^{N-1}}{\sqrt{2}}\right]^2}{\frac{\Delta^2}{12}}\right) = 10\ \log\left(2^{2N}\right) + 10\log\left(1.5\right)$$

$$= 6.02N + 1.76 \tag{9}$$

Med utgangspunkt i (9) vil maksimale SNR for 16 og 24 bits oppløsning, tilsvarende henholdsvis CD- og DVD-lyd, bli som følger:

- SNR 16bit = 98.1 dB
- SNR 24bit = 146.2 dB

## 2.3 Oversampling

Nyquist-Shannons samplingsteorem sier at sampleraten for en AD/DA må være minimum dobbelt så stor som den høyeste frekvenskomponenten som skal gjengis eller samples.

Hvis sampleraten er over dobbelt så høy som denne, vil vi ha *oversampling*.

I forbindelse med oversampling er det vanlig å definere over-sampling-ratio (OSR) som følger:

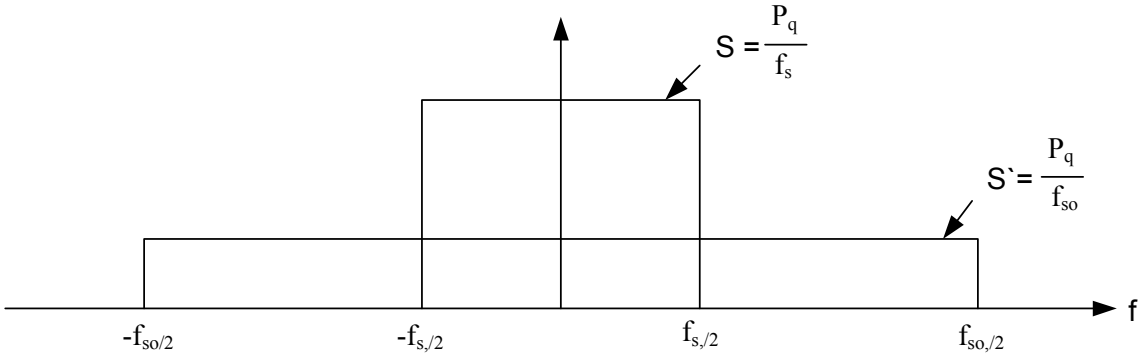
$$OSR = \frac{f_s}{2f_0} \tag{10}$$

Der $f_s$ er samplefrekvens (samplerate) og alle frekvenskomponenter av interesse ligger i basisbåndet lavere enn $f_0$.

### 2.3.1 Oversampling og SNR

Det viser seg at oversampling kan benyttes for å oppnå et høyere SNR enn det likningene i (8) og (9) skulle tilsi.

Grunnen til dette er at kvantiseringsstøyeffekten ved oversampling vil spres over et større frekvensbånd, samtidig som den totale kvantiseringsstøyeffekten vil være uforandret ettersom denne ikke er en funksjon av samplefrekvensen. Figur 8 illustrerer dette, der $P_q$ er den totale kvantiseringsstøyeffekten og S og S` er støytettheten ved samplefrekvensene $f_s$ og $f_{so}$.




**Figur 8. Sammenheng mellom støytetthet og samplefrekvens.**

Med utgangspunkt i uttrykket i (7) for den totale kvantiseringsstøyeffekten kan vi nå finne et uttrykk for kvantiseringsstøyen innenfor et gitt frekvensområde $f_{min}$ til $f_{max,}$ der vi antar at den totale støyeffekten er uniformt fordelt over hele Nyquist-båndet som i Figur 8:

$$P_q = \int_{f_{min}}^{f_{maks}} \frac{\Delta^2}{12} \cdot \frac{2}{f_s} \cdot \mathrm{d}f = \frac{\Delta^2}{12} \cdot \frac{2}{f_s} \cdot (f_{maks} - f_{\min}) \qquad \textbf{(11)}$$

Kvantiseringsstøyeffekt i basisbåndet mellom 0 og $f_0$ vil dermed tilsvare:

$$P_q = \frac{\Delta^2}{12} \cdot \frac{2}{f_s} \cdot f_0 = \frac{\Delta^2}{12} \cdot \frac{1}{OSR} \qquad \textbf{(12)}$$

Med utgangspunkt i (9) og (12) kan vi nå finne maksimalt SNR for et oversamplet signal med sinusform:

$$SNR_{maks} = N \cdot 6.02 + 1.76 + 10 \cdot log(OSR) \quad \textbf{(13)}$$

Ved å sammenlikne (9) og (13) ser vi at kvantiseringsstøyen reduseres med faktoren OSR ved oversampling, eller sagt på en annen måte:

- SNR forbedres med 3 dB for hver dobling av samplefrekvens.

Det er altså mulig å kompensere for et lite antall bit i en DA-konvertering ved å øke samplefrekvensen. Dette er meget interessant da det i praksis er vanskelig å realisere en DA-konverter som skal kunne gi ut for eksempel $2^{24}$ nøyaktige spenningsnivåer som tilfellet er ved 24-bits konvertering.

## *2.4 Oversampling med støyforming i delta-sigma-modulator*

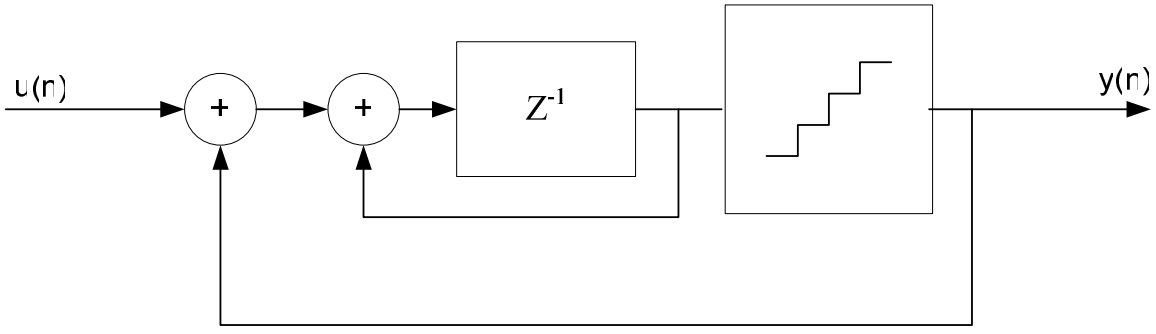
Vi har sett at det ved hjelp av oversampling er mulig å forbedre SNR utover det bit-bredden N på det digitale signalet skulle tilsi.

Det viser seg likevel at det ofte i seg selv ikke er tilstrekkelig å bare benytte oversampling for å oppnå det ønskede SNR, da dette kan kreve for høy samplefrekvens i forhold til det som er praktisk å realisere. Hvis man for eksempel vil oppnå CD-kvalitet ved 1-bit DA-konvertering, som er aktuelt i dette prosjektet, vil dette fra utrykket i (13) kreve en samplefrekvens på $44100 \cdot 2^{30}$=47352 GHz!

Det vil nå bli vist at det ved å kombinere oversampling med støyforming i en delta-sigma-modulator (DS) er mulig å forbedre det dynamiske området for en DA-konverter betraktelig.

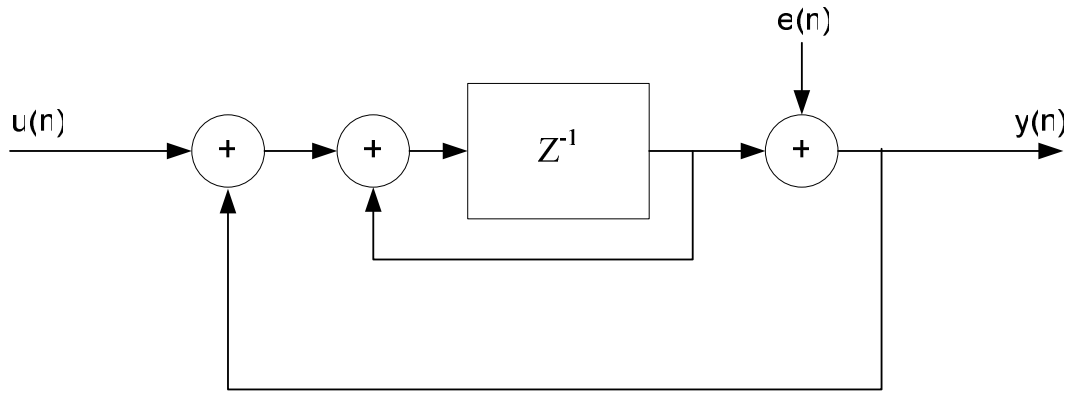
### 2.4.1 Prinsipp

Figur 9 viser blokkskjema for en 1. ordens DS-modulator med støyforming.




**Figur 9. 1. ordens delta-sigma-modulator.**

Figur 10 viser en 1. ordens delta-sigma-modulator der kvantisereren er byttet ut med modellen for denne som ble presentert i Figur 5.




**Figur 10. 1. ordens delta-sigma-modulator med modell for analyse av kvantiseringsstøy.**

## 2.4.2 Overføringsfunksjon

Med utgangspunkt i Figur 10 kan vi sette opp overføringsfunksjonene y/u, og y/e, som heretter vil bli omtalt som henholdsvis *signal transfer function* ($S_{TF}$) og *noise transfer function* ($N_{TF}$):

$$S_{TF}(Z) = \frac{Y(Z)}{U(Z)} = \frac{\frac{1}{(Z-1)}}{1+\frac{1}{Z-1)}} = Z^{-1} \quad \textbf{(14)}$$

$$N_{TF}(Z) = \frac{Y(Z)}{E(Z)} = \frac{1}{1+\frac{1}{Z-1)}} = 1 - Z^{-1} \quad \textbf{(15)}$$

Vi ser altså at signalet u(n) vil slippe uforandret gjennom systemet, bare forsinket med $z^{-1}$, mens kvantiseringsstøyen e(n) vil bli filtrert gjennom funksjonen $1\text{-}z^{-1}$, som tilsvarer et lavpassfilter. Frekvensresponsen for $N_{TF}$ er som følger:

$$N_{TF}(\omega) = N_{TF}(Z)\Big|_{Z=e^{j\omega}} = 1 - e^{-j\omega} = e^{-\frac{1}{2}j\omega}\left(e^{\frac{1}{2}j\omega} - e^{-\frac{1}{2}j\omega}\right)$$

$$= e^{-\frac{1}{2}j\omega} \cdot 2j \cdot \left( \frac{1}{2j} e^{\frac{1}{2}j\omega} - \frac{1}{2j} e^{-\frac{1}{2}j\omega} \right) \quad \textbf{(16)}$$

$$= j \cdot e^{-j\omega} \cdot 2 \sin\left(\frac{1}{2}\omega\right)$$

som mer praktisk kan uttrykkes som:

$$N_{TF}(\mathrm{f}) = |N_{TF}(\omega)| \Big|_{\omega = \frac{2\pi \mathrm{f}}{f_s}} = 2 \sin\left(\frac{\pi f}{f_s}\right) \quad \textbf{(17)}$$

Ved oversampling vil bare den nedre delen av frekvensspektret være interessant med tanke på kvantiseringsstøy. Vi er derfor interessert i å finne hvor mye kvantiseringsstøyen blir dempet i basisbåndet mellom 0 og $f_0$. Dette vil tilsvare middelverdien for $N_{TF}^2$ i denne regionen. I utregningen nedenfor antas det at signalet er oversamplet slik at $f_s >> f_0$, som gjør at vi kan anta at $\sin(\pi f/f_s) \approx \pi f/f_s$.

Dempingen av kvantiseringsstøy i basisbåndet $A_{basisbånd}$ som funksjon av OSR blir med dette som følger:

$$A_{basisbånd}(f_s) = \frac{\int_0^{f_0} \left[ 2 \frac{\pi f}{f_s} \right]^2 \mathrm{d}f}{f_0} = \frac{4\pi^2 f_0^2}{3 f_s^2} \quad \textbf{(18)}$$

$$A_{basisbånd}(OSR) = A_{basisbånd}(f_s) \Big|_{f_s = 2 f_0 OSR} = \frac{\pi^2}{3} \cdot \frac{1}{OSR^2} \quad \textbf{(19)}$$

eller

$$A_{basisbånd}(OSR) = 10 \log\left(\frac{\pi^2}{3}\right) - 20 \log(OSR) \quad \textbf{(20)}$$

$$= 5.17 - 20\,log\,(OSR)$$

Kvantiseringsstøy i basisbåndet vil altså bli betraktelig dempet i DS-modulatoren.

### 2.4.3 SNR etter støyforming

Vi kan nå finne den resulterende SNR ved å slå sammen bidraget fra oversamplingen med bidraget fra DS-modulatoren, som ble gitt i henholdsvis (20) og (13):

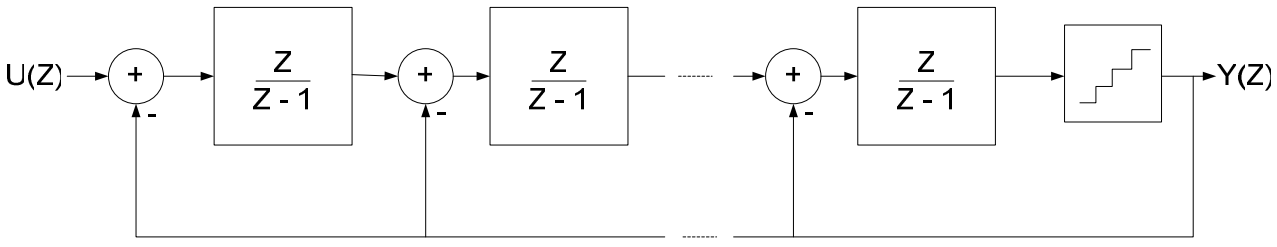
$$SNR_{maks} = 6.02N + 1.76 + 10\,log\,(OSR) - 5.17 + 20\,log\,(OSR) \quad \textbf{(21)}$$

$$= 6.02N - 3.41 + 30\,log\,(OSR)$$

### 2.4.4 Høyere ordens modulatorer

Vi har i (21) funnet et uttrykket for maksimalt SNR med en 1. ordens delta-sigma-modulator. I mange tilfeller er det ikke tilstrekkelig å bruke en 1. ordens modulator da dette fremdeles ville stille for store krav til OSR i forhold til det som er praktisk å realisere. Løsningen på dette kan være å benytte en høyere ordens DS-modulator. Spesifikasjonene for dette prosjektet definert i 1.1 vil som vi senere får se kreve en 5. ordens modulator. Ved valg av modulator-orden er det viktig å være oppmerksom på at modulatorens stabilitetsegenskaper vil ha sammenheng med denne. Detter er nærmere omtalt 2.4.5.

Figur 11 viser strukturen for en generell høyere ordens DS-modulator bestående av flere integratorer i kaskade [1].




**Figur 11. Generell struktur for høyere ordens DS-modulator.**

### 2.4.5 Stabilitet

Det kan vises at RMS-verdien for $N_{TF}$ over hele båndet fra 0 til $0.5f_s$ alltid vil være lik 1 [16]. Dette vil si at kvantiseringsstøyen egentlig ikke blir filtrert bort, men heller bare flyttet til et annet frekvensområde – derav støyforming. Høy demping av kvantiseringsstøy i basisbåndet vil gi tilsvarende høyere kvantiseringsstøy utenfor basisbåndet.

En DS-modulator er en tilbakekoplet sløyfe med et sterkt ulineært element, nemlig kvantisereren. Dette gjør at modulatoren har potensiale til å bli ustabil. Ved ustabilitet vil inngangssignalet til kvantisereren overskride det maksimale inngangsnivået for denne, noe som vil gi en kvantiseringsfeil større enn 1/2 Δ. Et resultat av dette kan være at modulatoren begynner å oscillere ukontrollert.

Det viser seg at sannsynligheten for ustabilitet øker med en aggressiv støyforming, det vil si når mye kvantiseringsstøy flyttes ut av basisbåndet.

Det finnes i dag ingen kjente metoder for å analytisk analysere stabilitetsegenskapene til en høyere ordens modulator. For å minske sannsynligheten for ustabilitet, er det utviklet diverse statistiske metoder for dette [27][28], i tillegg til at et modulatordesign bør gjennomgå omfattende simuleringer før man kan anse modulatoren som stabil. Det er også vanlig å bygge inn funksjonalitet som detekterer ustabilitet og nullstiller modulatoren hvis dette skulle inntreffe.

Det finnes også diverse anbefalinger og "tommelfingerregler" som bør følges for å minske sannsynligheten for ustabilitet i modulatoren. Av disse bør Lees regel nevnes. Denne sier at absoluttverdien til modulatorens $N_{TF}$ ikke bør overstige 1.5 for noen frekvenser ved 1-bit kvantisering.

Denne regelen kan være et utgangspunkt for dimensjonering av en modulator med flere bits kvantisering. Et høyere antall bits kvantisering vil bedre modulatorens lineære egenskaper, og det kan da tillates en noe mer aggressiv støyforming enn anbefalt av Lees regel.

Lees regel er ikke matematisk fundamentert, og det er anbefalt for et hvert modulatordesign å foreta omfattende simuleringer for å avdekke eventuell ustabilitet. En mangel ved regelen er at den ikke sier noe om nivået på inngangssignalet for DS-modulatoren. Ustabilitet blir initiert av overflyt i kvantisereren og vil derfor ha nær sammenheng med dette nivået. På bakgrunn av dette kan det være interessant å finne et utrykk som beskriver denne sammenhengen

Fra (14) og (15) kan vi sette opp følgende uttrykk for utgangssignalet fra DS-modulatoren:

$$Y(Z) = U(Z) \cdot S_{TF} + E(Z) \cdot N_{TF} \tag{22}$$

som i tidsplanet vil tilsvare

$$y(n) = u(n) * s_{tf}(n) + e(n) * n_{tf}(n) \quad \textbf{(23)}$$

Fra før vet vi at det er naturlig å anta $|s_{tf}(n)|=1$. Den maksimale verdien utgangssignalet y(n) kan innta vil dermed være gitt som:

$$max\,|\,y(n)\,| = max\,|\,u(n)\,| + max\,|\,e(n)\,| \cdot \|\,n_{tf}(n)\,\| \quad \textbf{(24)}$$

der

$$\|\,n_{tf}(n)\,\| = \sum |\,n_{tf}(n)\,| \quad \textbf{(25)}$$

$S_{TF}(Z)$ for en enkel flere-ordens modulatorstruktur som i Figur 11 vil tilsvare $(1\text{-}Z^{-1})^{M}$, der M er modulatororden. Det kan vises at $\|n_{tf}(n)\|$ som funksjon av M med dette kan uttrykkes som [1]:

$$\|\,n_{tf}(n)\,\| = 2^{M} \quad \textbf{(26)}$$

Toppnivået for u(n) og y(n) vil tilsvare det maksimale inngangsnivået for kvantisereren. Ved N-bits kvantisering kan vi anta at dette nivået tilsvare $2^{N}\Delta$, mens max|e(n)| vil tilsvare $\Delta$[1]. Med utgangspunkt i dette kan vi finne det maksimale nivået for inngangssignalet u(n) som kan tillates uten å risikere overflyt i kvantisereren:

$$max\,|\,u(n)\,| \le max\,|\,y(n)\,| - max\,|\,e(n)\,| \cdot \|\,n_{tf}(n)\,\|$$

$$\le \Delta \cdot 2^{N} - \Delta \cdot 2^{M} \quad \textbf{(27)}$$

som mer beskrivende med $u_{rel}$ kan uttrykkes relativt til det maksimale signalnivået $2^{N}\Delta$:

$$max\,|\,u_{rel}(n)\,| \le \frac{2^{N} - 2^{M}}{2^{N}} \quad \textbf{(28)}$$

[1] Vi antar her et system der inngangssignalet vi ligge mellom 0 og $2^{N}\Delta$ og der e(n) vil ligge mellom 0 og $\Delta$, ikke i regionen $\pm$ ½ $\Delta$ som antatt tidligere.

Uttrykket funnet i (28) er meget beskrivende for følgende grunnleggende sammenhenger i en delta-sigma-modulator:

- Det vil aldri kunne benyttes et fullskala inngangssignal uten å risikere overflyt i kvantisereren.
- En teoretisk grense for når det ikke kan oppstå overflyt i kvantisereren vil kun eksistere ved M<N.
- Sannsynligheten for ustabilitet øker med modulator-orden M.
- Sannsynligheten for ustabilitet minsker med antall bit kvantisering N.

Det er viktig å presisere at dette ikke er noen fullstendig stabilitetsanalyse. Det kan oppstå ustabilitet selv om inngangssignalet oppfyller (28) og modulatoren kan være stabil selv om inngangssignalet til tider overskrider dette nivået.

### 2.4.6 Idle tones

I forbindelse med delta-sigma-modulatorer kan det oppstå såkalte idle tones, det vil si uønskede frekvenser innenfor basisbåndet generert av DS-modulatoren selv. Disse tonene kan være hørbare og irriterende.

Det er i dette prosjektet valgt å ikke ta hensyn til dette fenomenet da det ikke er så interessant i en oppgave som dette, der sluttproduktet i seg selv strengt tatt ikke er det viktigste. Idle tones vil bare oppstå ved visse (lave) frekvenser, i tillegg til at det finnes trivielle metoder for å unngå disse. Skulle man derimot designe en DA-konverter for faktisk bruk eller salg, ville det selvsagt være nødvendig å vurdere denne problemstillingen nærmere.

Idle tones samt metoder for å unngå disse er nærmere beskrevet i [2].

## *2.5 Oversampling i praksis - Interpolering*

I en DA-konverter der man vil dra nytte av de nevnte egenskapene ved oversampling, må sampleraten til inngangssignalet økes, eller interpoleres, før en eventuell rekvantisering.

Interpolering med en faktor I kan utføres ved å sette inn I-1 nullverdier, såkalt ”zero-stuffing”, mellom hver sample i signalet som skal interpoleres. Matematisk kan dette uttrykkes som følger [3]:

$$v(m) = \begin{cases} x\left(\frac{m}{I}\right), & m = 0, \pm I, \pm 2I, \ldots \\ 0, & ellers \end{cases} \quad \textbf{(29)}$$

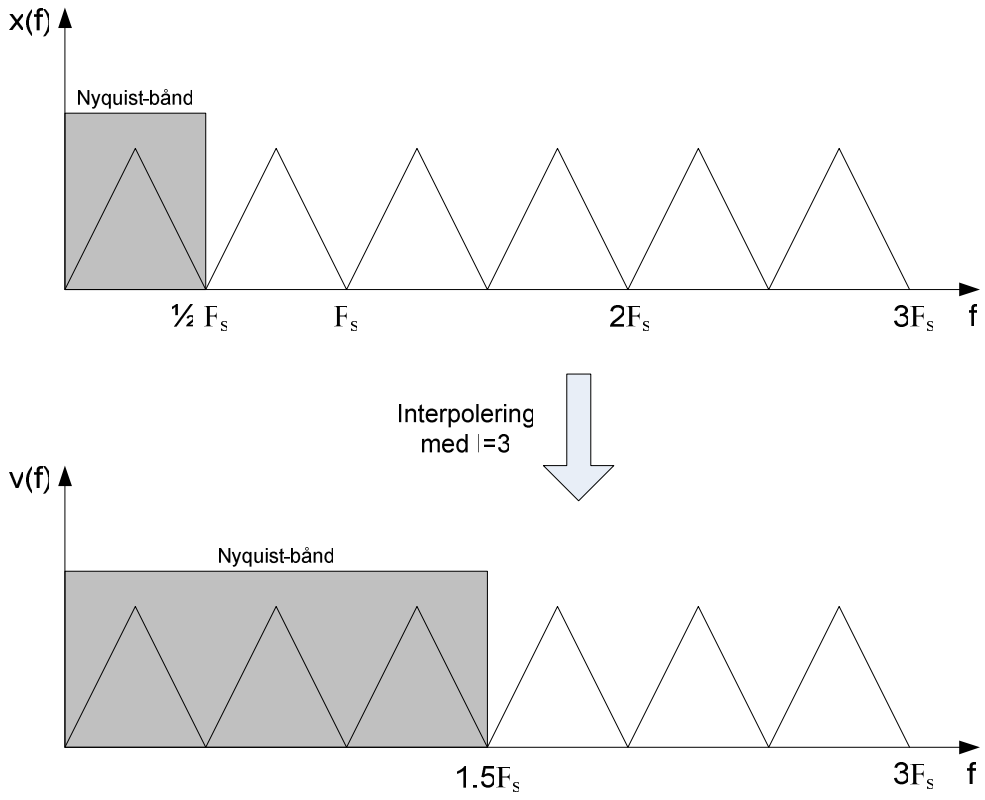
der x(n) er signalet før og v(m) er signalet etter interpoleringen.

### 2.5.1 Interpolering og aliasing

For et digitalt (samplet) signal vil alltid frekvensspekteret være symmetrisk rundt $nF_s$ for $n=\pm\infty$ [5], mens det bare er frekvenskomponenter lavere enn halve Nyquist-raten som vil kunne gjengis. Ved interpolering av et signal økes Nyquitst-raten, og frekvenskomponenter i det opprinnelige signalet som før ikke kunne gjengis vil nå opptre som såkalte aliasfrekvenser i Nyquist-båndet lavere enn halve Nyquist-raten.

Disse frekvenskomponentene i det interpolerte signalet er normalt uønsket og må filtreres bort i et digitalt interpoleringsfilter.

Figur 12 som viser et eksempel på aliasfrekvensene som oppstår i det interpolerte signalet v(n) etter interpolering av x(n) med faktor I=3.




**Figur 12. Frekvensinnhold i Nyquist-bånd før og etter interpolering.**

## 2.6 Interpoleringsfilter

For å fjerne de uønskede frekvenskomponentene i et interpolert signal benyttes det som regel konvensjonelle digitale lavpassfiltre. Ved design av slike filtre må man blant annet ta stilling til følgende hovedparametere:

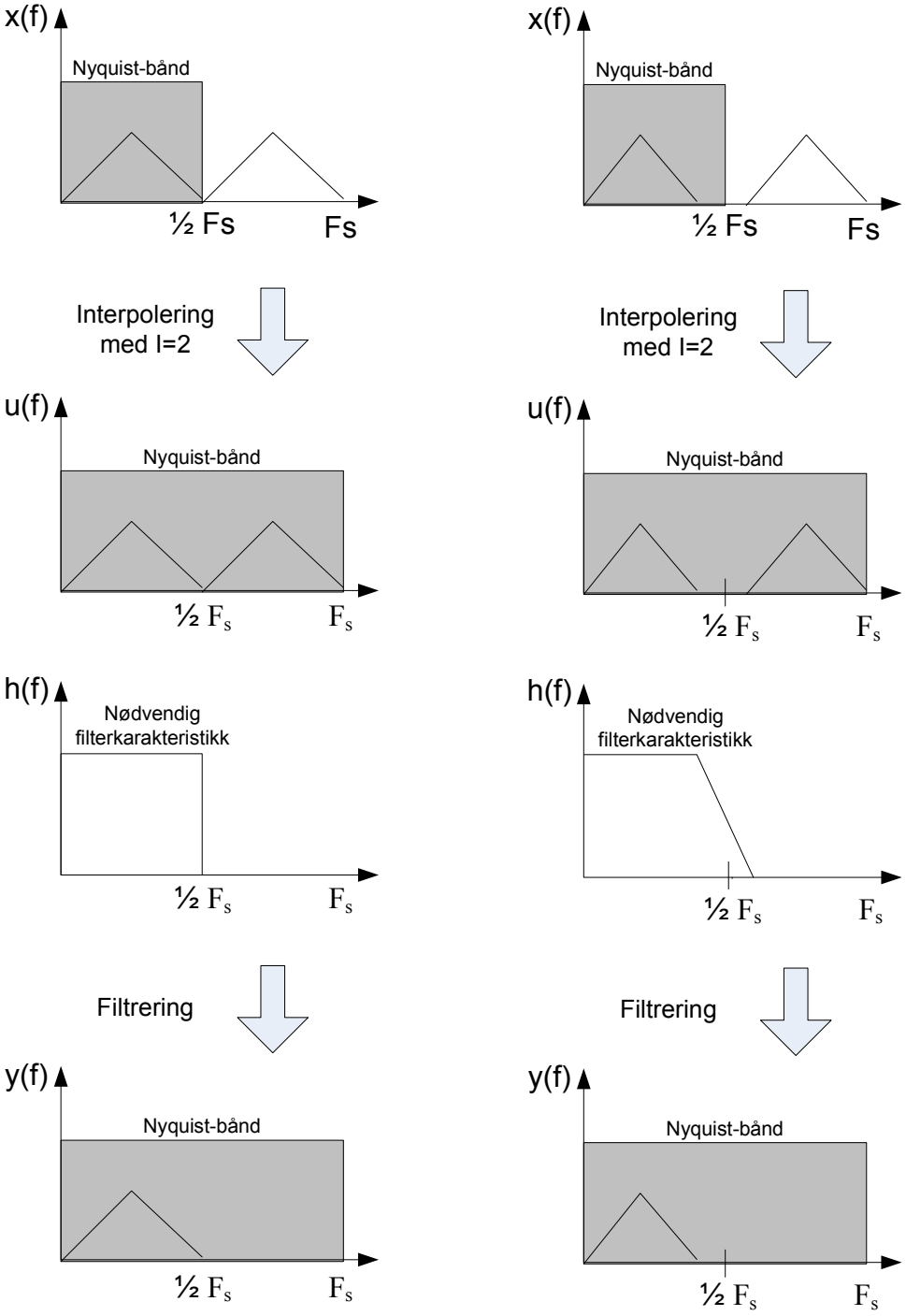
- Passbånd-rippel, det vil si variasjon av overføringsfunksjonen er i passbåndet.
- Stoppbånd-demping
- Bredde på transisjonsbånd

Transisjonsbåndet er overgangen mellom passbåndet og stoppbåndet. Den mulige bredden på transisjonsbåndet er gitt av hvor mye av informasjonen i Nyquist-båndet før interpolering vi ønsker å beholde også etter interpoleringsfilteret. Hvis vi vil beholde all informasjonen i signalet, altså hele Nyquist-båndet, vil dette kreve et filter med ekstremt kort transisjonsbånd, noe som er svært ressurskrevende å realisere.

Figur 13 illustrerer dette, der to signal interpoleres med faktoren I=2. Fra det ene signalet x(f) vil vi beholde all informasjonen i Nyquist-båndet, mens vi ved det andre signalet z(f) bare vil beholde informasjonen fra deler av Nyquist-båndet.

Vi ser at dette resulterer i høyere krav til interpoleringsfiltret for x(f) enn for z(f).

Transisjonsbåndet for interpoleringsfilteret vil altså være gitt av hvor stor del av Nyquist-båndet i inngangssignalet som inneholder informasjon av interesse. Denne problemstillingen tas det hensyn til ved definering av standarder for digital lyd. Eksempelvis er sampleraten for CD lik 44.1 kHz, noe som gir et Nyquist-bånd opp til 22.05 kHz, mens det hørbare området imidlertid bare regnes å være opp mot 20 kHz [33].




**Figur 13. Interpolering og krav til interpoleringsfilter.**

## 2.7 Filtertyper

### 2.7.1 FIR-filter

I et *Finite Impulse Response* (FIR) -filter beregnes filtrets utgangsverdier utelukkende på bakgrunn av tidligere inngangsverdier. Det kan vises at et FIR filter alltid vil være BIBO-stabilt (*Bounded Input – Bounded Output*), det vil si at utgangssignalet alltid vil være begrenset så lenge inngangssignalet er det.

Overføringsfunksjonen til et FIR filter er gitt av en mengde koeffisienter b(n). Sammenhengen mellom inngangssignalet x(n), koeffisientene b(n) og utgangssignalet y(n) er for et FIR-filter som følger [3]:

$$y(n) = b(n) \cdot x(n) + b(1) \cdot x(n-1) + \ldots + b(M) \cdot x(n-M) \quad \textbf{(30)}$$

som også kan uttrykkes som

$$y(n) = \sum_{i=0}^{M} b(i) \cdot x(n-i) \quad \textbf{(31)}$$

der M+1 er antallet koeffisienter i b(n).

Operasjonen mellom b(n) og x(n) i **(31)** kalles også convolution, og betegnes ofte med operatoren *.

Det er rett fram å ut fravise at impulsresponsen h(n) for filteret vil være identisk med filterkoeffisientene b(n).

### 2.7.2 IIR-filter

I et *Infinite Impulse Response* (IIR)-filter beregnes utgangsverdiene fra filteret på grunnlag av tideligere inngangsverdier *og* tidligere utgangsverdier. I motsetning til et FIR-filter er ikke et IIR-filter garantert BIBO-stabilt.

Overføringsfunksjonen til et FIR filter er gitt av koeffisientene b(n) og a(n). Sammenhengen mellom inngangssignalet x(n), koeffisientene b(n) og a(n) og utgangssignalet y(n) er for denne filtertypen som følger [3]:

$$y(n) = b(0)x(n) + b(1)x(n-1) + .. + b(M)\,x(n-M) + \\ a(0)x(n-1) + a(1)x(n-2) \ldots + a(Q)\,(n-Q-1) \tag{32}$$

som også kan uttrykkes som

$$y(n) = \sum_{i=0}^{M} b(i)x(n-i) + \sum_{k=0}^{Q} a(k)y(n-k-1) \tag{33}$$

der M+1 er antallet koeffisienter i b(n) og Q+1 er antallet koeffisienter i a(n).

### 2.7.3 Sammenlikning av FIR- og IIR-filter

Det vil nå bli foretatt en sammenlikning av de viktigste egenskapene for FIR- og IIR-filtre. Denne sammenlikningen kan brukes som utgangspunkt ved valg av filtertype i designet.

**Stabilitet**
Et FIR-filter vil alltid være BIBO-stabilt mens dette ikke er tilfelle for et IIR-filter.

**Kompleksitet**
Et FIR-filter vil generelt behøve mange flere filterkoeffisienter, altså beregninger, enn et IIR-filter med samme frekvensrespons vil behøve.

**Symmetrisk impulsrespons**
Symmetrisk impulsrespons kan utrykkes på følgende måte:

$$h(n) = \pm\, h(M-1-n), \qquad n = 0, \ldots, M-1 \tag{34}$$

Vi vet at impulsresponsen for et FIR-filter vil tilsvare filterkoeffisientene {B}, så det er altså fullt mulig å oppnå symmetrisk impulsrespons i FIR-filter. Dette er imidlertid ikke mulig i et IIR-filter [3].

Det kan vises at et filter med symmetrisk impulsrespons vil ha konstant gruppeforsinkelse, noe som igjen gir lineær faserespons [3]. Med dette menes at alle frekvenskomponenter i signalet vil forsinkes like mye gjennom filteret. Denne egenskapen er særlig ønskelig i forbindelse med audio [22].

Symmetriske koeffisienter er også fordelaktig med tanke på hardware-implementering, da to og to koeffisienter er like, noe som vil spare lagringsplass i ROM.

### 2.7.4 Halvbånd FIR-filter

I multiratesystemer benyttes det ofte halvbånd FIR-filter ved interpolering eller desimering med faktoren 2. Det viser seg at det er mulig å lage særlig effektive implementeringer av denne filtertypen, der tilnærmet halvparten av filterkoeffisientene vil være lik 0, noe som igjen innebærer at antallet beregninger halveres i forhold til det filterordenen skulle tilsi [30].

Halvbånd FIR-filter av denne typen vil ha følgende karakteristikk:

- Passbånd og stoppbånd like stort
- Transisjonsbånd er sentrert rundt midten av Nyquist-båndet.

Et slikt filter kan konstrueres ved å ta utgangspunkt i et FIR-filter med et odde antall N symmetriske koeffisienter $B_1(i)$, og modifisere disse på følgende måte [21][30]:

$$B_2(i) = \begin{cases} B_1\left(\frac{i}{2}\right), & n = 0,\ 2,\ 4,\ldots,\ N-1 \\ 1, & n = \left(\frac{N}{2}\right) - 1 \\ 0, & \textit{ellers} \end{cases} \quad \textbf{(35)}$$

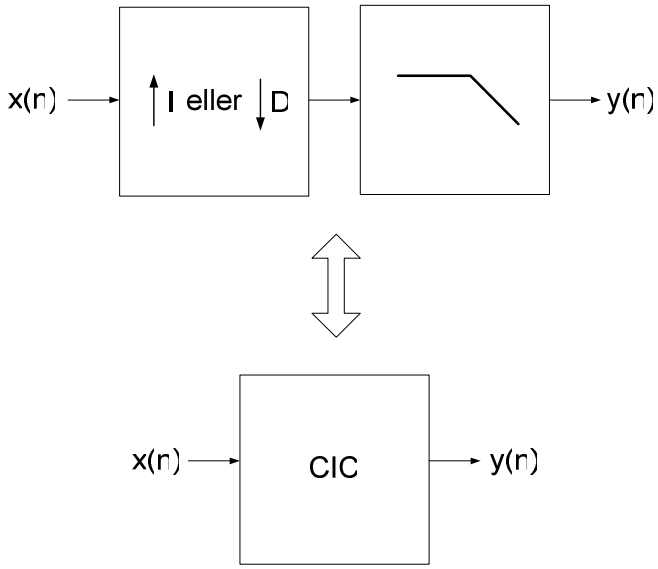
Det resulterende filteret gitt av koeffisientene $B_2(i)$ vil være et halvbåndfilter med en bredde på passbånd (og stoppbånd) lik den halve bredden av passbåndet for det modifiserte filteret.

Med metoden ovenfor vil det resulterende halvbåndfilteret få en forsterkning på 6 dB, som ofte er ønskelig da signalnivået i basisbåndet vil halveres ved en interpolering med I=2. Denne reduksjonen av signalnivå ved interpolering skyldes at halvparten av signalkomponentene i det interpolerte signalet vil ligge utenfor basisbåndet, samtidig som den totale signalenergien er uforandret da interpolering bare innebærer innsetting av 0-verdier.

### 2.7.5 CIC-filter

*Cascade integrator comb*, ofte omtalt som CIC-filter, er en teknikk som kombinerer samplerate-omforming (interpolering eller desimering) og det nødvendige påfølgende lavpassfilteret [6]. Denne teknikken behøver ingen multiplikasjoner og er derfor særlig godt egnet for hardware-implementering.

Figur 14 illustrere den ekvivalente funksjonen for et CIC-filter. Det vil videre bare bli fokusert på interpolering ved hjelp av denne teknikken, da det er dette som blir benyttet i denne oppgaven.



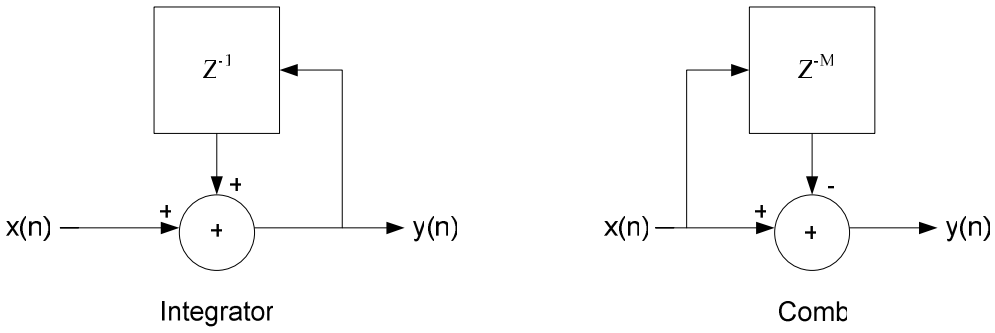

**Figur 14. Ekvivalent funksjon for CIC-filter.**

## Oppbygning

Et CIC-filter består av følgende to grunnelementer:

- Integrator
- ”Comb”

Figur 15 viser oppbygningen av disse.




**Figur 15. Grunnelementer i CIC-filter.**

Grunnelementene i Figur 15 er i prinsippet et IIR-filter og et FIR-filter som kan beskrives av følgende uttrykk:

$$y(n) = x(n) + y(n-1) \tag{36}$$

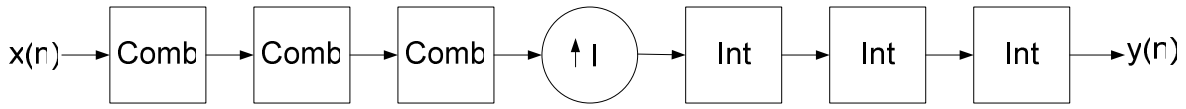
for en integrator og

$$y(n) = x(n) - x(n-M) \tag{37}$$

for en "comb"

Variabelen M kalles differensiel forsinkelse og settes normalt til 1 eller 2 [6].

Figur 16 viser et CIC-filter for interpolering med faktoren I som benytter N=3 integrator- og comb–enheter.




**Figur 16. Interpolernde CIC-filter med N=3 comb- og integratortrinn.**

## Overføringsfunksjon og frekvensrespons

Det kan vises at overføringsfunksjonen for det ekvivalente interpoleringsfilteret i et CIC-filter er som følger [6]:

$$H(z) = H_I(z)H_C(z) = \frac{(1-z^{-I\cdot M})^N}{(1-z^{-1})^N} = \left( \sum_{k=0}^{I\cdot M-1} z^{-k} \right)^N \tag{38}$$

Uttrykket i (38) viser at overføringsfunksjonen ekvivalent med N FIR-filtere i kaskade, alle med filterkoeffisienter lik 1. Et CIC-filter har altså symmetrisk impulsrespons og dermed lineær fase.

Fra overføringsfunksjonen i (38) kan det vises at frekvensresponsen for filteret er gitt som:

$$|H(f)| = \left| \frac{\sin(\pi M f)}{\sin\left(\pi \frac{M f}{I}\right)} \right|^N \quad \textbf{(39)}$$

**Forsterkning**

Det er mulig å se ut fra frekvensresponsen i (39) at et CIC-filter vil ha en betydelig forsterkning ved lave frekvenser. Dette gjør det som regel nødvendig å dempe utgangssignalet før det kan brukes videre. Med tanke på implementering i hardware er det også interessant å kjenne forsterkningen for hvert trinn utover i CIC-filteret, slik at registre og adderere skal kunne dimensjoneres deretter.

For en CIC interpolator er forsterkningen for hvert trinn gitt som følger:

$$G(i) = \begin{cases} 2^i, & i = 1,\ 2,\ldots,\ N \\ \dfrac{2^{2N-i}(I \cdot N)^{i-N}}{I}, & i = N+1,\ldots,\ 2N \end{cases} \quad \textbf{(40)}$$

Det bør nevnes at det ikke er mulig å dempe signalet for hvert trinn, da de små avvikene introdusert ved avrunding og forkorting kan føre til ustabilitet i integratorseksjonene i filteret [6].

## 2.8 1-bit DA og dynamisk feil

I dette prosjektet skal det analoge signalet genereres ved å filtrere det 1-bits utgangssignalet fra FPGA-kretsen i et eksternt analogt lavpassfilter.

Spesifikasjonene satt for prosjektet angir bare hva kvaliteten på det digitale signalet i FPGA-kretsen skal være, men det er likevel viktig å kjenne til hva som kan forringe signalkvaliteten ved denne analoge omformingen, og hvordan det digitale designet eventuelt kan tilpasses disse begrensningene.

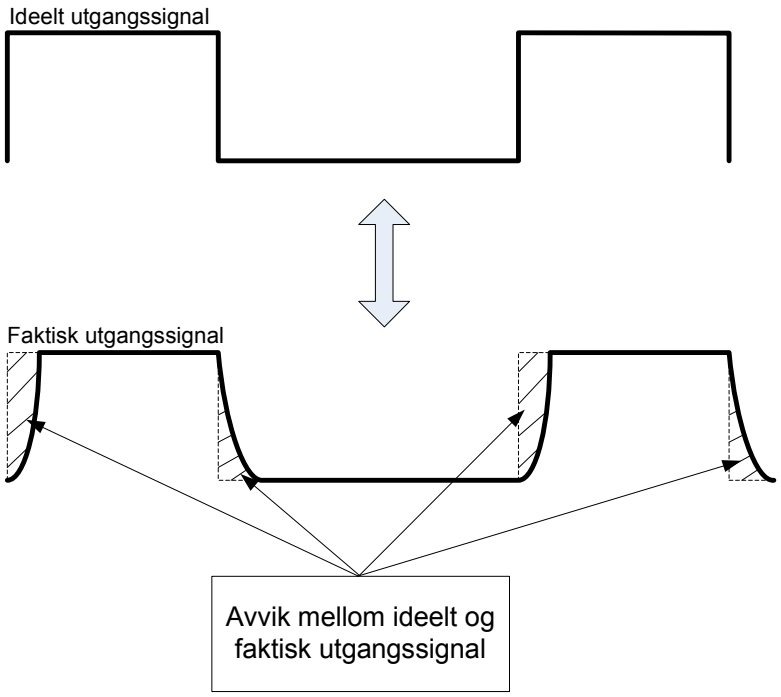
I denne sammenheng bør det spesielt tas hensyn til stige- og falltidsbegrensningene man alltid vil ha på utgangen av FPGA-kretsen og i eventuelle aktive komponenter i det analoge lavpassfilteret, samt jitter i det 1-bits utgangssignalet. Jitter skyldes klokkeunøyaktighet, som vil føre til at transisjonstidspunktene for utgangssignalet vil være ikke-ideelle.

Begge disse begrensningene vil gi avvik i signalenergien ved hver signaltransisjon, som illustrert i Figur 17 og Figur 18.

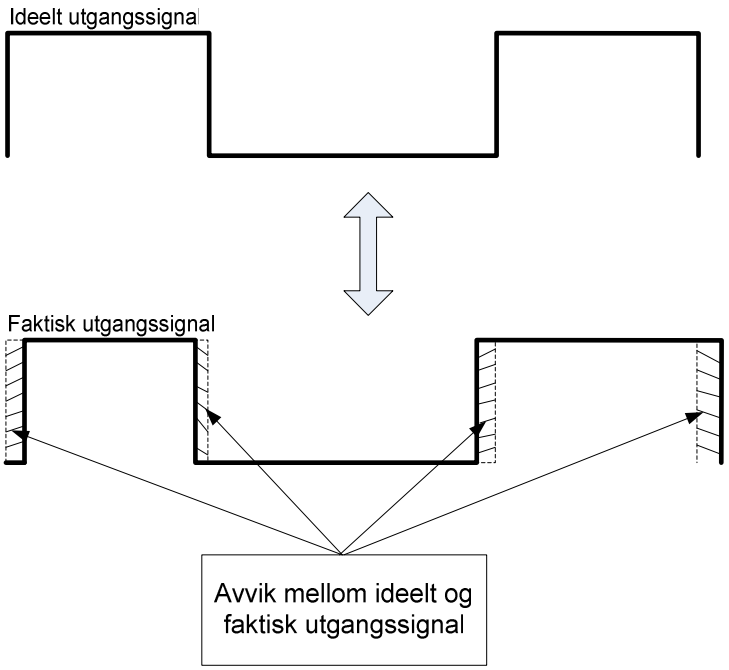
Ved vanlig *pulse-density-modulation* (PDM), det vil si der det analoge nivået representeres med en bit-strøm uten en fast pulsfrekvens, vil dette gi opphav til harmonisk forvrenging (THD) ettersom antallet og tidspunktene for transisjonene vil være signalavhengig.

I tillegg vil det også kunne oppstå *inter-symbol-interferens* (ISI), da avviket ved hver enkelt puls vil være avhengig av nivået til den foregående pulsen. Eksempelvis vil det ved to etterfølgende pulser med verdi 1 bare være et avvik ved stigende flanke på den første pulsen og fallende flanke på den andre.

På bakgrunn av dette er det intuitivt å slå fast at bit-strømmen som representerer signalnivået bør etterstrebe å ha et lavest mulig og fast antall transisjoner. Med dette oppnår man et mer signal-uavhengig avvik som bare vil være en funksjon av modulasjonsfrekvensen, og dermed tilsvare en DC-offset i basisbåndet (tilnærmet).




**Figur 17. Forandringer i utgangssignal som følge av stige- og falltidsbegrensninger.**

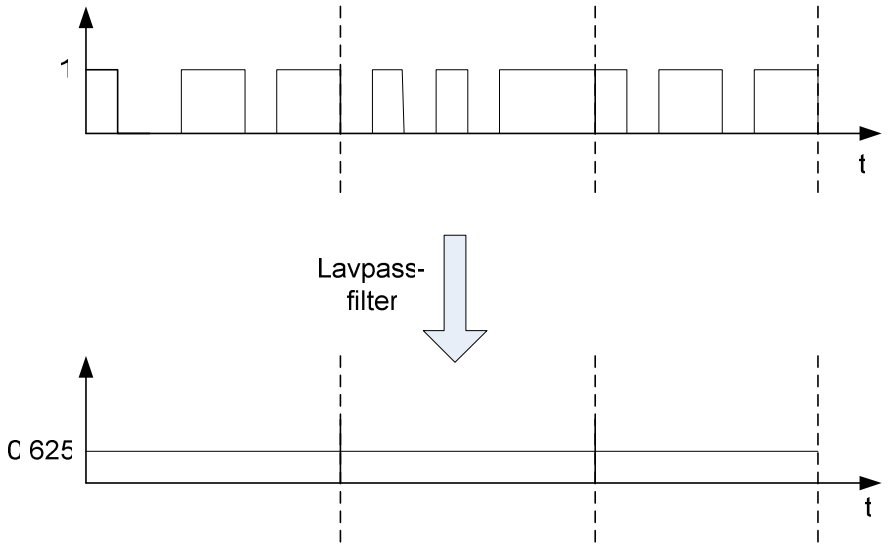



**Figur 18. Forandringer i utgangssignal som følge jitter.**

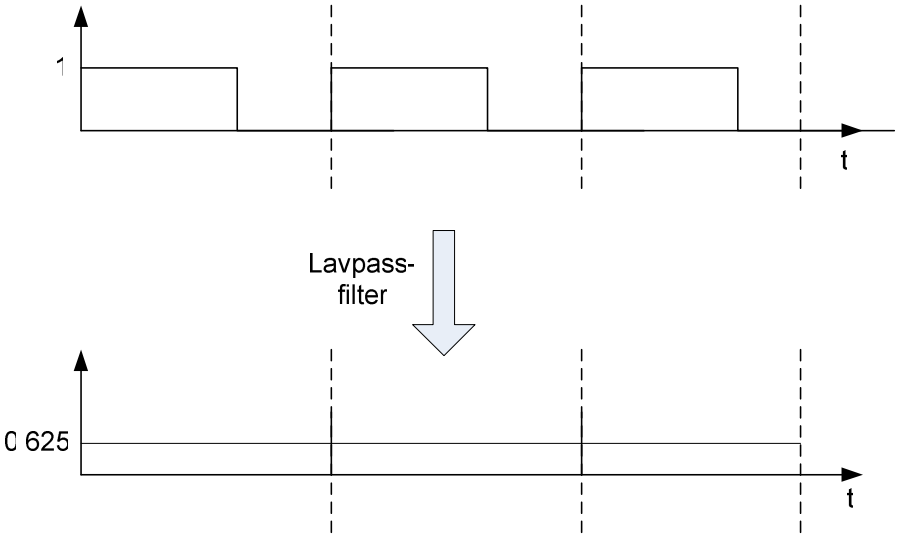
## 2.9 Pulsbredde-modulering (PWM)

Utgangssignalet fra en 1-bit delta-sigma DA-konverter vil tilsvare et PDM-signal, ettersom dette ikke vil ha en fast pulsfrekvens. I 2.8 så vi at dette er ugunstig da det vil medføre THD og ISI ved den analoge omformingen. En metode å begrense effekten av de ikke-ideelle egenskapene som er årsaken til dette kan være å benytte *pulsbredde-modulasjon* (PWM).

Ved PWM representeres det analoge utgangssignalet ved hjelp av pulsbredden på det digitale utgangssignalet, mens pulsfrekvensen holdes konstant. Dette gir et signaluavhengig antall transisjoner og dermed en reduksjon av problemene beskrevet i 2.8. Figur 19 viser et analogt nivå representert ved hjelp av PDM, mens Figur 20 viser det samme nivået presentert ved hjelp av PWM.




**Figur 19. Representasjon av signalnivå ved hjelp av PDM.**



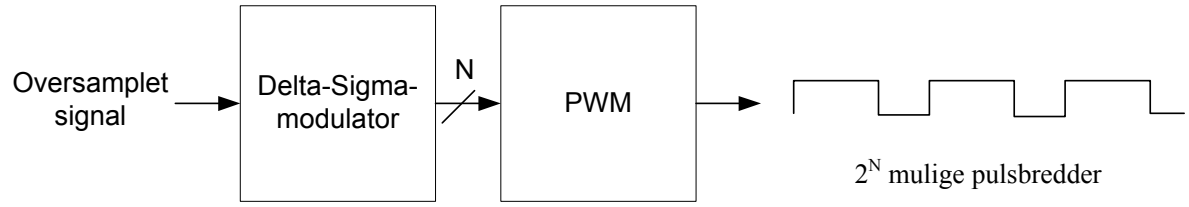

**Figur 20. Representasjon av signalnivå ved hjelp av PWM.**

I tillegg til at antallet transisjoner er fast ved PWM, ser vi også at antallet transisjoner vil være lavest ved denne modulasjonstypen, som vi i 2.8 fastslo er ønskelig.

Ved bruk av PWM i sammenheng med en DS-modulator vil utgangssignalet fra DS-modulatoren være et multibit signal som vist i Figur 21. Med PWM kan det altså benytte flere bits kvantisering, som vi vet i seg selv vil gi et bedre SNR, i tillegg til å være gunstig for DS-modulatorens stabilitetsegenskaper.

En ulempe med med PWM er imidlertid at PWM-modulatoren vil behøve en klokkefrekvens lik $2^N$ ganger samplefrekvensen ut fra DS-modulatoren for å kunne generere det pulsbredde-modulerte utgangssignalet. Da man vanligvis har en begrensning på den øvre klokkefrekvensen som kan benyttes, innebærer dette at OSR for systemet vil måtte halveres for hver bit bredden N til utgangen av DS-modulatoren økes. Fra (21) ser vi at dette vil redusere den resulterende SNR, da SNR er mer følsom for reduksjon av OSR enn for øking av bit-bredden. Utrykket i (21) gjelder også bare for en 1. ordens DS-modulator. Ved høyere ordens modulatorer vil følsomheten for reduksjon av OSR økes ytterligere.

Disse ulempene vil måtte veies opp mot fordelene ved innføring av PWM.



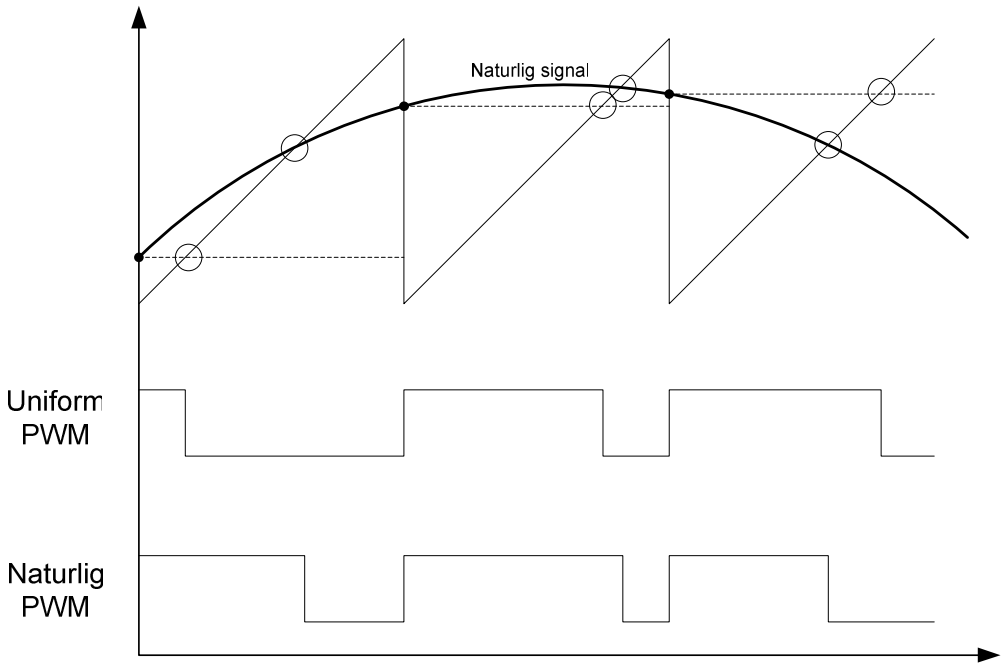

**Figur 21. Prinsipp for DS-modulator med PWM.**

### 2.9.1 PWM modulasjonstyper

Bulsbredde-modulsajon blir tradisjonelt delt inn i 2 hovedklasser; uniform PWM (UPWM) og naturlig PWM (NPWM). Hovedprinsippet for begge disse er at utgangssignalet genereres ved å sammenlikne inngangssignalet med et trekant- eller sagtannformet signal. Ved å skifte nivå på utgangssignalet hver gang nivåene på disse to signalene krysses, vil utgangssignalet tilsvare inngangssignalet pulsbredde-modulert med en moduleringsfrekvens lik frekvensen på det trekantformede signalet [18].

Forskjelle mellom de to hovedklassene består i hvorvidt det trekant- eller sagtannformede signalet sammenliknes med en sample av inngangssignalet, som ved UPWM, eller med det kontinuerlige inngangssignalet, som ved NPWM.

Figur 22 viser prinsippet for disse to modulasjonstypene.




**Figur 22. Signal representert ved hjelp av uniform PWM og naturlig PWM.**

Vi ser at det modulerte utgangssignalene vil være forskjellig for de to modulasjonstypene. Det viser seg at dette gir seg utslag i harmonisk forvrengning ved UPWM, som blir nærmere beskrevet i 2.9.2.

Figur 22 viser enkeltsidig PWM der det brukes et sagtannformet signal for sammenlikning. Hvis det brukes en trekantformet puls kalles det dobbeltsidig PWM. Fordelen med sistnevnte er at man da vil halvere den effektive frekvensen på utgangssignalet, uten å redusere informasjonsmengden i signalet.

### 2.9.2 Frekvenskomponenter i et PWM signal

Ved PWM vil det i tillegg til det ønskede signalet genereres en rekke forskjellige støykomponenter. Analyser av NPWM viser at frekvensspekteret på utgangen vil inneholde følgende frekvenskomponenter [11]:

1) Inngangssignalet
2) PWM svitsjefrekvensen
3) harmoniske av svitsjefrekvens
4) miksing mellom inngangssignalet og de harmoniske av svitsjefrekvensen

Ved UPWM vil man i tillegg til de nevnte frekvenskomponentene få:

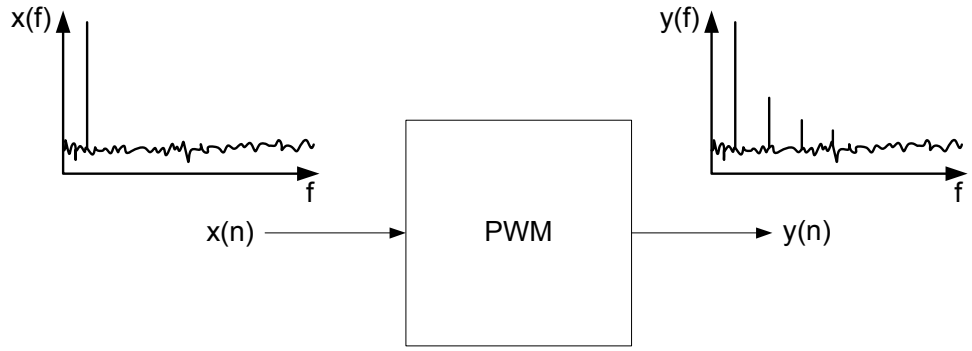
5) Harmoniske av inngangssignalet

Komponent 1 er åpenbart ønskelig å ha i utgangssignalet, mens komponent 2 vil ligge utenfor basisbåndet og i så måte ikke være et problem. De resterende komponentene kan oppstå i

basisbåndet og føre til økning av THD+N. Ifølge [11] vil frekvenskomponent 5 være betydelig større enn 3 og 4 for praktiske moduleringsfrekvenser, det vil si moduleringsfrekvenser mye høyere enn signalfrekvensen. For eksempel vil bidrag 3 og 4 ved en moduleringsfrekvens på 10 ganger signalfrekvensen gi en støy på -144 dB i basisbåndet, altså ned mot støygulvet for 24 bits kvantisering.

NPWM er åpenbart ikke mulig å realisere digitalt da dette forutsetter et kontinuerlig inngangssignal til PWM-modulatoren, så støybidraget i punkt 5 vil altså være det viktigste å begrense.

Figur 23 illustrerer problemet med overharmoniske signalkomponenter i forbindelse med UPWM. Matematiske metoder for å illustrere dette kan finnes i [11] og [18].




**Figur 23. PWM og harmonisk støy.**

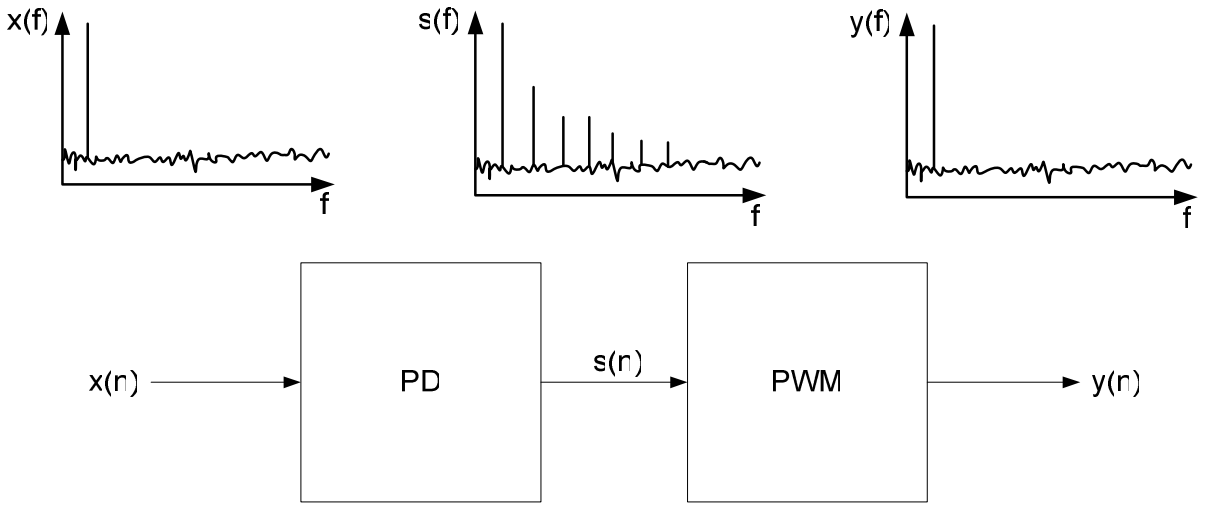
### 2.9.3 Reduksjon av overharmoniske ved UPWM

Hovedutfordringen ved bruk av UPWM er å redusere de overharmoniske signalkomponentene i basisbåndet mest mulig. Vi vet at disse uønskede signalkomponentene skyldes at krysningspunktet mellom sample-verdien inn til modulatoren og det modulerende trekantsignalet ikke blir korrekt i forhold til det ideelle krysningspunktet man får ved NPWM, som vist i Figur 22.

En metode for å redusere dette problemet er å erstatte sampleverdiene inn til modulatoren med verdier som gir krysningspunkter nærmere de ideelle. Dette innebærer at det kontinuerlige signalet mellom hver sample må estimeres på grunnlag av samplene fra inngangssignalet.

Denne modifiseringen av signalet kalles predistortion (PD), da det faktisk tilføres støy til signalet før pulsbredde-modulatoren.

Figur 24 illustrerer prinsippet for redusering av overharmoniske signalkomponenter ved hjelp av predistortion.




**Figur 24. Prinsipp for reduksjon av harmoniske ved hjelp av *predistortion*.**

### 2.9.4 Oppsummering av PWM

Vi har sett på fordelene og ulempene ved bruk av pulsbredde-modulering. I utgangspunktet kan det se ut som det er langt flere ulemper enn fordeler ved bruk av PWM. Innføring av PWM vil både øke designets kompleksitet i form selve modulatoren i tillegg til at det kan være behov for å benytte predistortion.

Fordelen ved PWM er at dette kan minske inter-symbol-interferens og harmonisk forvrengnig i det analoge utgangstrinnet samt at det kan benyttes multibit DS-modulering, noe som er gunstig med tanke på stabilitet og THD.

Hovedgrunnen til at PWM viser seg hensiktsmessig å benytte i mange tilfeller, er rett og slett at det er lettere å oppnå denne forbedringen ved å ta høyde for dette i det digitale designet, enn det vil være å oppnå den samme forbedingen ved å modifisere det analoge utgangstrinnet.

# 3 Design av delta-sigma DA-konverter

*Det vil i dette kapitelet bli gitt en gjennomgang av designprosessen for DA-konverteren i denne oppgaven.*

## 3.1 Fremgangsmåte

All dimensjoneringen og valg av kretsparametere er hovedsaklig gjort ved hjelp av modellering i Matlab. Først etter at et en tilfredsstillende systemmodell i Matlab forelå, ble de laget en implementering for hardware ved hjelp av Verilog. Deretter ble hardware-implementeringen verifisert ved at simuleringsdata fra denne ble analysert i Matlab og sammenliknet med simuleringsdata fra det modellerte Matlab-systemet.

Denne typen analyse av hardware-implementeringen har vært særlig interessant da den vil vise alle faktiske avvik mellom implementeringen og den modellerte Matlab-modellen.

## 3.2 Litt om spesifikasjonene

Konstruksjon av en delta-sigma DA-konverter krever forståelse av en rekke grunnleggende parametere og sammenhenger for å kunne foreta de avveiningene som er nødvendig for å få et vellykket design.

Utgangspunktet er som oftest et sett kravspesifikasjoner, som både vil omhandle krav til ytelse og begrensninger for hvor kompleks den resulterende hardwaren kan bli.

I 1.1 ble det gitt en beskrivelse av spesifikasjonene som er satt for dette prosjektet. Vi kan repetere disse:

- SNR > 120 dB
- THD+N < -100 dB
- Designet skal ha et 1-bits utgangssignal for lavpassfiltrering i eksternt lavpassfilter

Som vi ser inneholder spesifikasjonene bare krav til ytelse, og ingen hardware-begrensninger.

For det videre arbeidet i oppgaven har det likevel vært nødvendig å også definere et mål for hvilke hardware-begrensninger designet skal oppfylle. Konstruksjon av en DA-konverter uten denne typen begrensninger ville for det første være uinteressant, da dette ville være en triviell oppgave, men ikke minst urealistisk da man alltid vil ha visse begrensninger når noe skal realiseres i hardware.

Disse begrensningene for denne oppgaven er definert med utgangspunkt i det planlagte utviklingsoppsettet bestående av en AK4117 SP-DIF dekoder for innhenting av data fra digital lydkilde, samt en Xilinx Spartan-3 XC3S200 FPGA for implementering.

Når det gjelder AK4117 vil denne i tillegg til å levere det digitale inngangssignalet til DA-konverteren også lever de nødvendige klokkesignalene til denne. Disse omfatter samplefrekvensen inn $F_s$, samt frekvensen $512F_s$ som kan benyttes for klokking av raske prosesser synkronisert med $F_s$. For high-end ytelse må konverteren kunne gjengi lyd med CD-kvalitet, noe som vil tilsvare henholdsvis 44.1 kHz og 22.6 MHz for de to klokkesignalene.

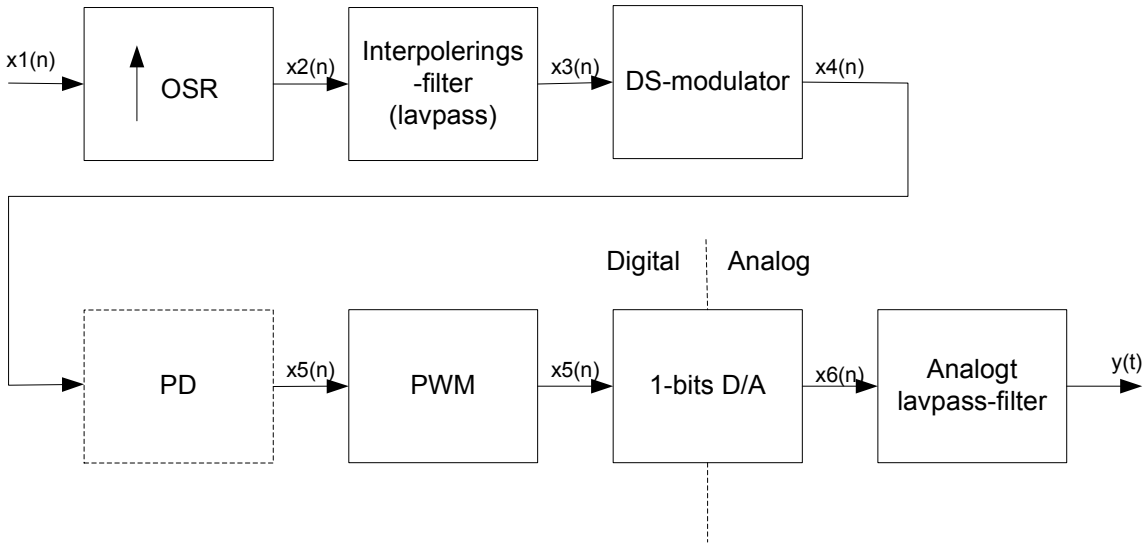
Med tanke på XC3S200 har et naturlig mål vært å kunne implementere og teste systemet på denne. Det er ikke satt noen spesifikasjoner for strømforbruk.

Vi kan oppsummere disse hardware-begrensningene:

- Kretsen kan maksimalt ha en klokkefrekvens på $512F_s$, der $F_s$ er sampleraten for inngangssignalet.
- Kretsen må kunne brukes med en klokkefrekvens $512F_s$=22.6 MHz.
- Kretsen bør kunne implementeres på en Xilinx XC3S200 FPGA.

## 3.3 Systemtopologi

Figur 25 viser topologien som har vært utgangspunktet for DA-konverteren. Dette er en tradisjonell topologi for en delta-sigma DA-konvertere med et pulsbreddemodulert 1-bit utgang. Alternativt kunne man velge å ikke bruke PWM, men som beskrevet i 2.8 vil dette kunne føre til økning av THD ved den analoge omformingen.




**Figur 25. Valgt toplogi for delta-sigma DA-konverter.**

I figuren er predistortion-modulen (PD) angitt med stiplet linje da det er vanskelig å på forhånd avgjøre om denne enheten er nødvendig for å møte de gitte spesifikasjonene. Denne avgjørelsen kan først tas etter at andre systemparametere er fastsatt og det er foretatt simuleringer med disse.

## 3.4 Dimensjonering av DS-modulator

Det er hensiktsmessig å starte med dimensjonering av DS-modulatoren da parametrene for denne vil være nødvendig for dimensjonering de andre modulene i systemet.

De aktuelle parametrene ved dimensjonering av DS-modulatoren er som følger:

- Over-sampling-ratio (OSR)
- Antall kvantiseringsnivåer N
- Modulator-orden
- Plassering av nullpunkter for *noise-transfer-function* ($N_{TF}$)

### 3.4.1 Sammenheng mellom OSR og antall kvantiseringsnivåer

Vi kan først fastslå at den øvre grensen for OSR vil være gitt av den maksimale klokkefrekvensen for systemet gitt som 512 ganger sampleraten til inngangssignalet. Maksimal verdi for OSR vil altså være lik 512.

Fra utrykket i (13) så vi at det er ønskelig med en høy OSR da dette vil gi best SNR.

Samtidig vet vi fra avsnitt 2.9 at pulsbreddemodulatoren krever en klokkefrekvens på $2^N$ ganger sampleraten ut fra DS-modulatoren, der $2^N$ er antallet kvantiseringsnivåer i denne. Sampleraten på utgangen av DS-modulatoren vil være lik $OSR \cdot F_s$, og med en maksimal klokkefrekvens på $512F_s$ får man følgende sammenheng:

$$OSR \cdot F_s \leq 512 \cdot F_s \tag{41}$$

som kan forenkles til

$$OSR \cdot F_s \leq 512 \cdot F_s \tag{42}$$

og som også kan uttrykkes med hensyn på det maksimale antall bit kvantisering N:

$$N_{maks} = \frac{ln\left(\frac{512}{OSR}\right)}{ln(2)} \tag{43}$$

Altså vil en lav OSR gi flere mulige kvantiseringsnivåer, noe vi fra 2.4.5 vet vil være positivt for modulatorens stabilitetsegenskaper. Lav OSR er også gunstig med tanke på implementeringen i hardware, da det med dette vil kunne utføre flere operasjoner i hardware for hver sample, noe som muliggjør deling av hardware-ressurser.

Vi kan altså konkludere med at det er ønskelig med en så lav OSR som mulig, forutsatt at spesifikasjonene for SNR overholdes.

For det videre modulator-designet er det valgt å benytte en OSR på 32, noe som vil gi 16 kvantiseringsnivåer og mulighet til å utføre beregninger over 16 klokkeperioder for hver utgangsverdi.

### 3.4.2 Om modulatororden

Høyere ordens modulator gir bedre SNR, men øker også designets kompleksitet og gir større sansnylighet for ustabilitet i modulatoren.

### 3.4.3 Om plassering av nullpunkter

Det er mulig å benytte strukturer for DS-modulatoren som gir stor frihet i valg av $N_{TF}$. Ved hjelp av koeffisienter kan det være mulig å velge ønsket plassering av poler og nullpunkter i overføringsfunksjonen uavhengig av hverandre. Det er ikke lett å si noe generelt om den optimale formen eller de optimale egenskapene for $N_{TF}$, da dette vil variere fra design til design.

Lavt signal-støy-forhold (SNR) er uansett en egenskap som er ønskelig i de fleste applikasjoner. Det viser seg at ved å fordele nullpunktene i $N_{TF}$ jevnt utover basisbåndet, vil frekvensresponsen for $N_{TF}$ få en flatere karakteristikk i basisbåndet. Dette vil gi en lavere RMS-verdi for $N_{TF}$ i dette båndet[1]. Optimalisering av nullpunktene på denne måten vil altså kunne gi et forbedret SNR.

En modulatorstruktur som tillater plassering av nullpunktene på denne måten vil kreve noen flere beregninger enn en struktur uten denne muligheten.

### 3.4.4 Om valg av parametere

Parametrene for DS-modulatoren er valgt på bakgrunn av simuleringer av forskjellige modulator-oppsett i Matlab. Ved dimensjonering av DS-modulatoren er man altså primært interessert i modulatorens $N_{TF}$. Som nevnt finnes det en rekke forskjellige modulator-strukturer, men det er på dette tidspunkt ikke nødvendig å ta stilling til hvilken struktur som skal brukes da denne senere kan tilpasses den valgte $N_{TF}$.

For generering av $N_{TF}$ er det benyttet *The Delta-Sigma Toolbox 7.1* for *Matlab* [31]. Dette verktøyet genererer en passende $N_{TF}$ for ut fra hovedparametrene OSR, modulator-orden, nullpunktplassering og maksimal forsterkning utenfor basisbåndet.

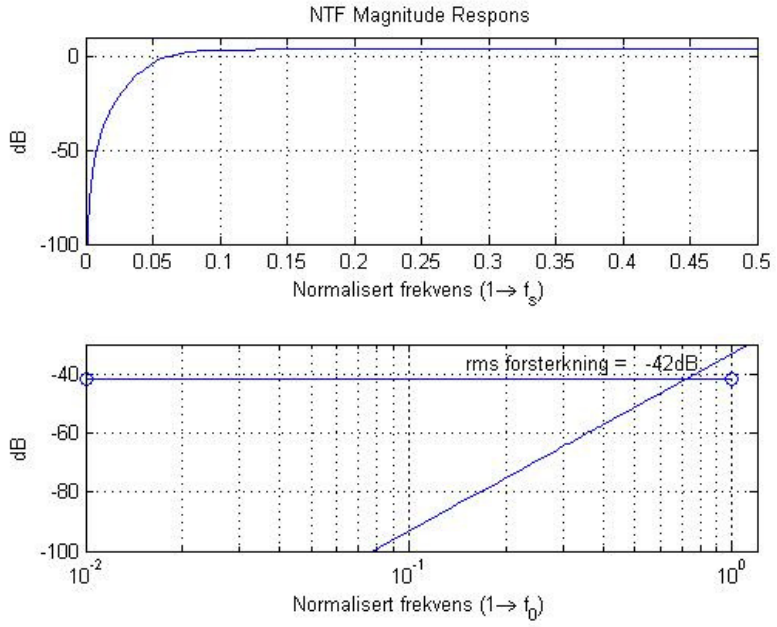
Det vil i 3.4.5 til 3.4.8 bli presentert et utvalg av simuleringene som har vært grunnlaget for valg av DS-modulatoren i denne oppgaven. Vi vil her se hvordan $N_{TF}$ påvirkes av forandringer i de nevnte modulatorparametrene.

### 3.4.5 Simulering DS-oppsett 1

Som et utgangspunkt kan vi foreta simuleringer med følgende parametere:

- OSR=32
- Modulator-orden=3
- Maksimal forstrekning=1.5
- Ikke optimaliserte nullpunkter

Figur 26 viser overføringsfunksjonen generert på grunnlag av disse parametrene evaluert over hele Nyquist-båndet til det oversamplede utgangssignalet. Figuren viser også et utsnitt av basisbåndet fra f=0 $\rightarrow f_0$, altså det båndet vi vil flytte kvantiseringsstøy ut av.




**Figur 26. $N_{TF}$ overføringsfunksjon DS-oppsett 1.**

Vi ser at $N_{TF}$ har en RMS-forsterkning på -42 dB i basisbåndet. Kvantiseringsstøyen vil altså bli redusert med 42 dB i dette båndet.

På bakgrunn av (13) kan vi beregne maksimalt SNR med dette modulatoroppsettet:

SNR=N*6.02+1.76+10*log(OSR)+42 ≈ 83 dB

Et SNR på 83 dB er ikke tilfredsstillende i forhold til de gitte spesifikasjonene.

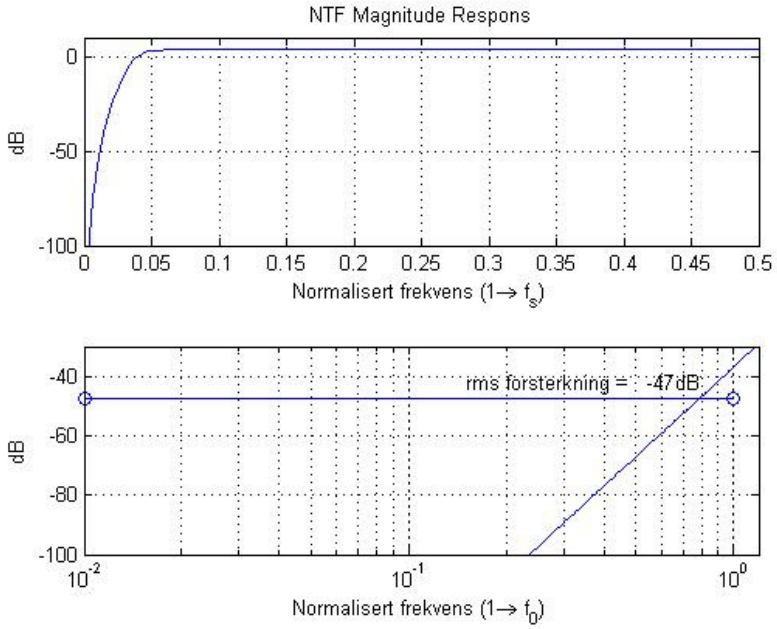
### 3.4.6 Simulering DS-oppsett 2

Da SNR ved forrige oppsett ikke var tilfredsstillende, kan vi nå forsøke å forandre modulatororden fra 3 til 5.

Parametrene for 2. simulering blir altså:

- OSR=32
- **Modulator-orden=5**
- Maksimal forstrekning=1.5
- Ikke optimaliserte nullpunkter

Figur 27 viser resulterende $N_{TF}$ med dette modulatoroppsettet.




**Figur 27. NTF overføringsfunksjon DS-oppsett 2.**

Vi ser at reduksjon av kvantiseringsstøy i basisbåndet og dermed SNR øker med 5dB ved forandring av modulatororden fra 3 til 5.

Maksimalt SNR blir med dette lik 86dB + 5dB = 88 dB.

Høyere modulatororden gir som forventet bedre SNR, men vi ser at SNR fremdeles er for lavt i forhold til de gitte spesifikasjonene.

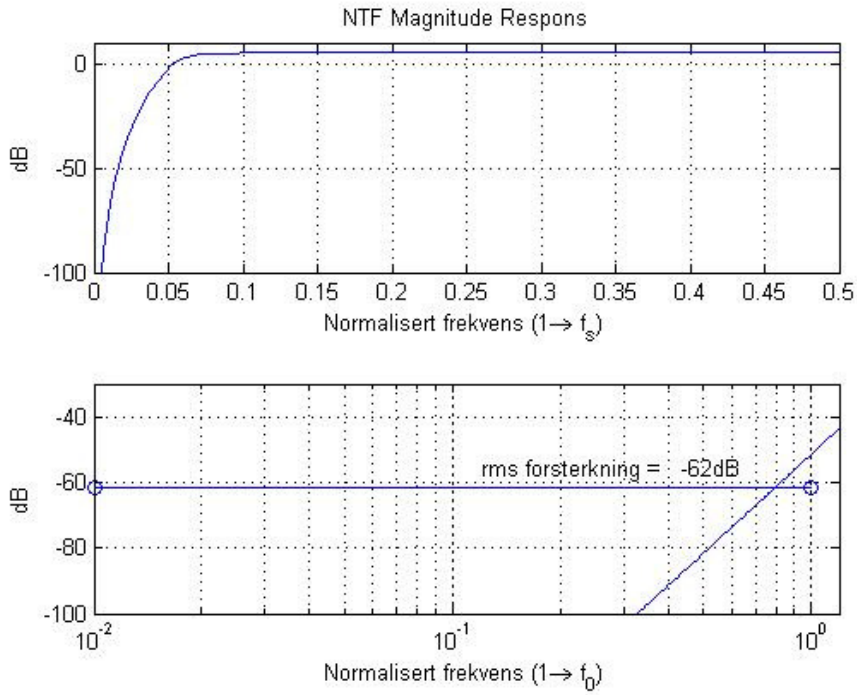
### 3.4.7 Simulering DS-oppsett 3

I simuleringene så langt har den maksimale forsterkningen for $N_{TF}$ med bakgrunn i Lees regel vært satt til 1.5. Anbefalingen om en maksimal forsterkning på 1.5 gjelder imidlertid for 1-bit kvantisering. Siden vi her har 4-bits kvantisering, vil det nok kunne tillates en noe mer aggressiv støyforming. Faktisk anbefalte Lees regel opprinnelig en maksimal forsterkning lik 2, før denne grensen senere ble moderert til 1.5 [1]. Det kan derfor være rimelig å anta at det er "trygt" med en forsterkning på 1.8 ved 4-bits kvantisering.

Parametrene for 3. simulering blir altså:

- OSR=32
- Modulator-orden=5
- **Maksimal forstrekning=1.8**
- Ikke optimaliserte nullpunkter

Figur 28 viser $N_{TF}$ med dette modulatoroppsettet.




**Figur 28. NTF overføringsfunksjon DS-oppsett 3.**

Vi ser at reduksjon av kvantiseringsstøy i basisbåndet, og dermed SNR, øker med 15 dB ved forandring av den maksimale forsterkningen fra 1.5 til 1.8. Altså at vil selv en liten økning av forsterkningen utenfor basisbåndet gi en betydelig reduksjon av kvantiseringsstøy i basisbåndet. Dette burde ikke være uventet da vi vet at arealet under grafen for NTF alltid vil

være konstant lik 1, samtidig som basisbåndet bare tilsvarer en liten del (1/OSR) av hele Nyquist-båndet.

Maksimalt SNR blir med dette lik 91 dB + 15 dB = 103 dB.

Dette er fremdeles ikke tilstrekelig i forhold til de gitte spesifikasjonene for SNR.

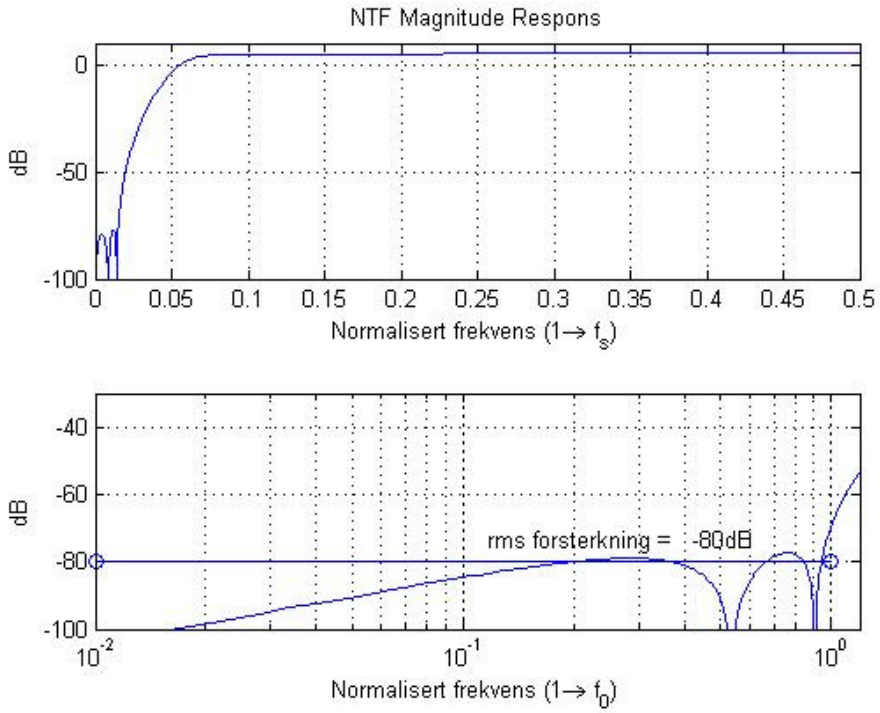
### 3.4.8 Simulering DS-oppsett 4

Simuleringene å langt er bare foretatt med modulatoroppsett der alle nullpunkter for $N_{TF}$ er plassert i f=0. Vi kan derfor foreta en simulering med optimaliserte nullpunkter, altså der nullpunktene for $N_{TF}$ er jevnt fordelt utover basisbåndet.

Parametrene for 4. simulering blir altså:

- OSR=32
- Modulatororden=5
- Maksimal forstrekning=1.8
- **Optimaliserte nullpunkter**

Figur 29 viser $N_{TF}$ med dette modulatoroppsettet.




**Figur 29. NTF overføringsfunksjon DS-oppsett 4.**

Fra Figur 29 ser vi at optimalisering av nullpunktene fører til en flatere $N_{TF}$ i basisbåndet, som øker RMS-verdien for reduksjon av kvantiseringsstøy fra 62 til 80 dB.

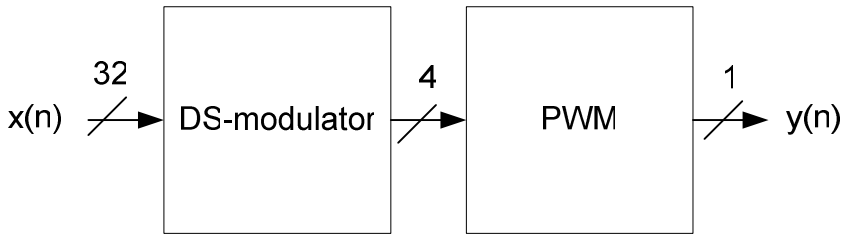
Maksimalt SNR blir med dette lik 106 dB + 18 dB = 121 dB.

Dette modulatoroppsettet kan derfor være utgangspunkt for den videre dimensjoneringen av DA-konverteren.

## 3.5 PWM og predistortion

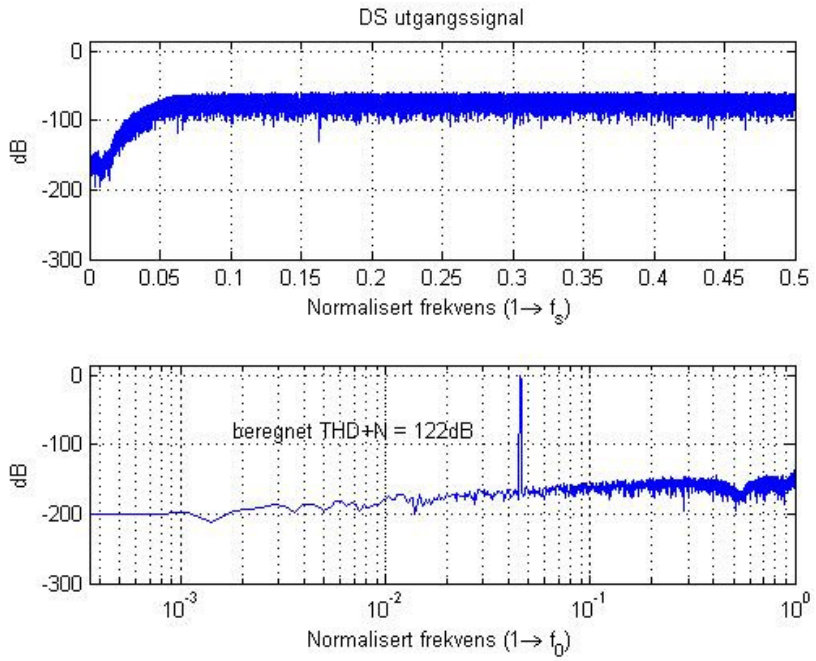
Vi kan nå foreta en simulering av det valgte oppsettet for DS-modulatoren med og uten PWM. Oppsettet for denne simulering er illustrert i Figur 30. Inngangssignalet x(n) som tilføres DS-modulatoren er et sinussignal med amplitude -1dBr, 0 dBFS, og en diskret frekvens på $1/(22 \cdot f_0 \cdot OSR)$, som vil tilsvarer et 1 kHz CD-signal etter oversampling. Inngangssignalet har her 32-bits oppløsning for å hindre at kvantiseringsstøy i inngangssignalet påvirker simuleringsresultatet. Det er her heller ikke her tatt hensyn til DS-modulatorens struktur.

Fra denne simuleringen vil vi kunne verifisere at oppsettet for DS-modulatoren er hensiktsmessig samt kunne avgjøre om det er nødvendig med predistortion for å begrense de overharmoniske frekvenskomponentene som vil tilføres ved PWM.

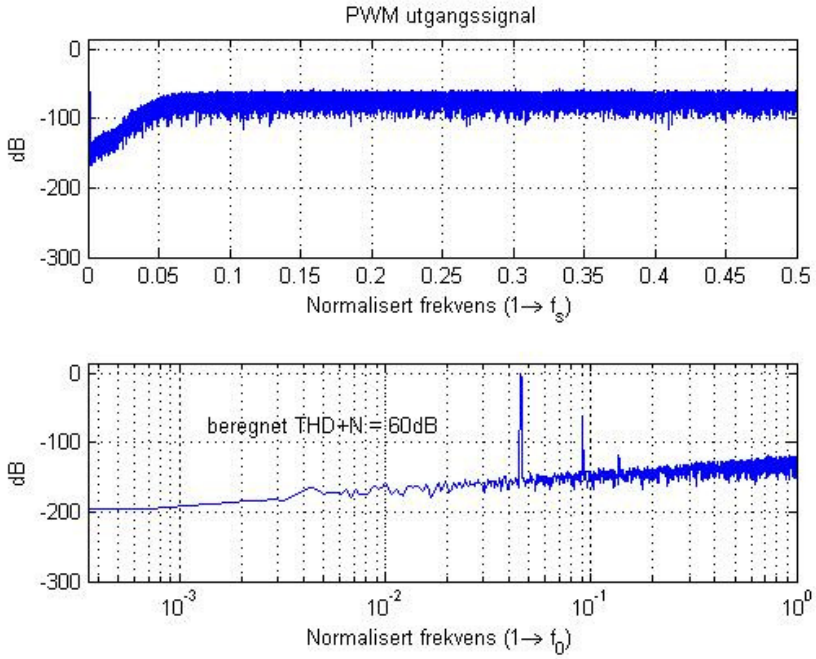



**Figur 30. Simuleringsoppsett for DS-modulator med PWM.**

Figur 31 viser frekvensspekteret på utgangen av DS-modulatatoren mens Figur 32 viser utgangssignalet y(n) fra PWM-modulatoren.




**Figur 31. Utgangssignal DS-modulator.**




**Figur 32. PWM utgangssignal.**

I Figur 31 ser vi at utgangssignalet fra DS-modulatoren har en THD+N på -122 dB, som er innenfor det man kunne forvente ut fra beregningen for SNR i 3.4. (Det er for øvrig en fortegnsfeil ved angivelse i THD+N i figurene, denne skal være negativ).

Videre ser vi i Figur 32 at THD+N blir betydelig dårligere etter PWM-modulatoren, med en økning til 60 dB. Vi ser tydelig at dette skyldes harmoniske frekvenskomponenter som tilføres signalet i PWM-modulatoren.

Vi kan konkludere med at det er nødvendig med en eller annen form for predistortion for å forbedre THD+N.

### 3.5.1 Valg av algoritme for predistortion

Det er utviklet og beskrevet en rekke forskjellige algoritmer for konvertering av et uniformt samplet PCM-signal til et naturlig PWM-signal. Det ville være ønskelig å analysere et utvalg av disse algoritmene for å kunne velge den optimale algoritmen for dette prosjektet, men dette har imidlertid ikke blitt gjennomført da det ville være en uhensiktsmessig omfattende jobb.

Det blir i [10] presentert 2 slike algoritmer, der de presenterte algoritmene også sammenliknes opp mot en rekke andre kjente algoritmer. Disse sammenlikningene viser at de presenterte algoritmene har svært god ytelse i forhold til algoritmenes kompleksitet.

Tabell 1 viser THD det i [10] hevdes man kan oppnå med de presenterte algoritmene. Det er for øvrig verdt å nevne at disse verdiene gjelder for en OSR lik 8, mens det for DA-konverteren i denne oppgaven er valgt en OSR lik 32. Med en høyere samplerate vil man nok kunne forvente en ytterligere forbedring av THD ettersom dette vil være "nærmere" ideell kontinuerlige sampling [8].

| | **THD [dB]** | | | | |
|---|---|---|---|---|---|
| | 1 kHz | 4 kHz | 5 kHz | 6.6 kHz | 10 kHz |
| **Algoritme A** | -136.39 | -101.47 | -95.61 | -88.48 | -78.43 |
| **Algoritme B** | -143.00 | -119.25 | -114.98 | -113.07 | -103.55 |

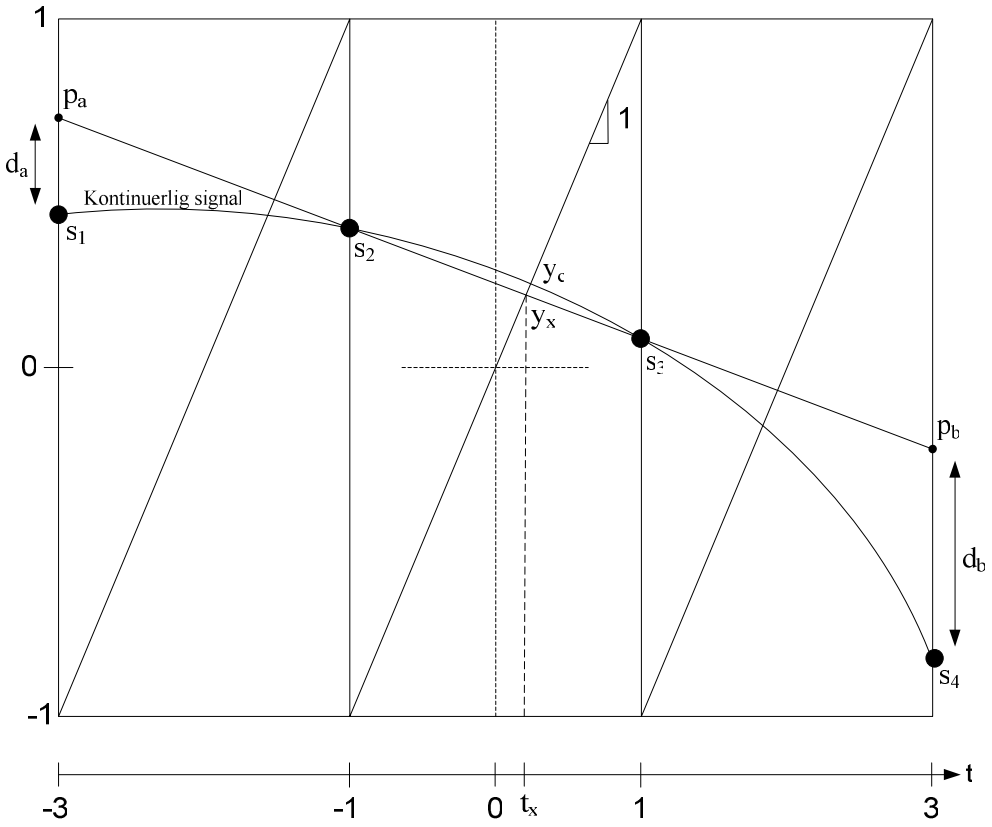
**Tabell 1. Spesifikasjoner for predistortion-algoritmer presentert i [10]. Spesifikasjonene gjelder ved OSR=8 og signalamplitude 90% av maksimal amplitude.**

Vi ser av Tabell 1 at algoritme B har noe bedre ytelse enn algoritme A, men at begge algoritmene synes å være god nok for å overholde spesifikasjonene i dette prosjektet.

Siden algoritme A krever en enklere implementering er det valgt å ta utgangspunkt i denne ved det videre designet.

### 3.5.2 Beskrivelse av valgt predistortion-algoritme

Algoritme A kan enklest beskrives med utgangspunkt i Figur 33.




**Figur 33. Prinsipp for *predistortion*-algoritme presentert i [10].**

Hensikten med algoritmen er å estimere krysningspunktet $y_c$ mellom det kontinuerlige (naturlige) og det sagtannformede modulasjonssignalet. Dette estimatet beregnes på grunnlag av de 4 samplene $s_1$-$s_3$ av det kontinuerlige signalet.

Først konstrueres den rette linjen $p_a$-$p_b$ gjennom $s_2$ og $s_3$. Deretter finnes krysningspunktet $y_x$= $t_x$ mellom denne linjen og det sagtannformede signalet. Det endelige krysningspunktet $y_c$ blir estimert ved å modellere en parabolsk signalform mellom $s_2$ og $s_3$, der krumningen på signalet bestemmes av differansene $d_a$= $d_d$-$s_1$ og $d_b$= $p_b$-$s_3$. Krysningspunktet $y_c$ gis deretter av følgende uttrykk:

$$y_c = y_x + \alpha \cdot \left( d_a + d_d \right) \cdot \left( 1 - t_x^2 \right) \qquad \textbf{(44)}$$

Algoritmen kan også uttrykkes ved hjelp av *pseudo*-kode på følgende måte:

```
1. tx=((1/2)*(s1+s2))/(1-(1/2)*(s2-s1))
2. yx=tx
3. da=(2*s2)-s3
4. db=(2*s3)-s2
5. yc=yx+ α*(da+dd)*(1-tx^2)
```

Konstanten α må finnes empirisk. Hvis vi tenker oss et sinusformet inngangssignal, vil α være optimal for en enkel signalfrekvens. Denne sammenhengen er likevel ikke så sterk at den forhindrer oss i å finne en verdi for α som vil være tilfredsstillende for alle audio-frekvenser [10].

### 3.5.3 Beregning av koeffisient α

I [10], der algoritme A er beskrevet, blir det ikke oppgitt passende verdier for koeffisienten α. Det blir imidlertid beskrevet en metode for å finne denne, som innebærer å simulere algoritmen med en mengde forskjellige verdier av α, for så å velge den verdien som gir det beste resultatet.

I denne oppgaven er det imidlertid valgt en mer analytisk tilnærming for å løse dette.

Uttrykket i (44) viser funksjonen for å finne det estimerte krysningspunktet $y_c$. Vi kan uttrykke denne mer generelt til å gjelde det estimerte signalet $y_e$ mellom $s_2$ og $s_3$ på følgende måte:

$$y_e(t) = y_l(t) + \alpha \cdot \left(d_a + d_d\right) \cdot (1 - t^2) \tag{45}$$

der $y_l(t)$ er funksjonen for den rette linjen mellom $s_2$ og $s_3$.

Vi kan videre uttrykke (45) med hensyn på α:

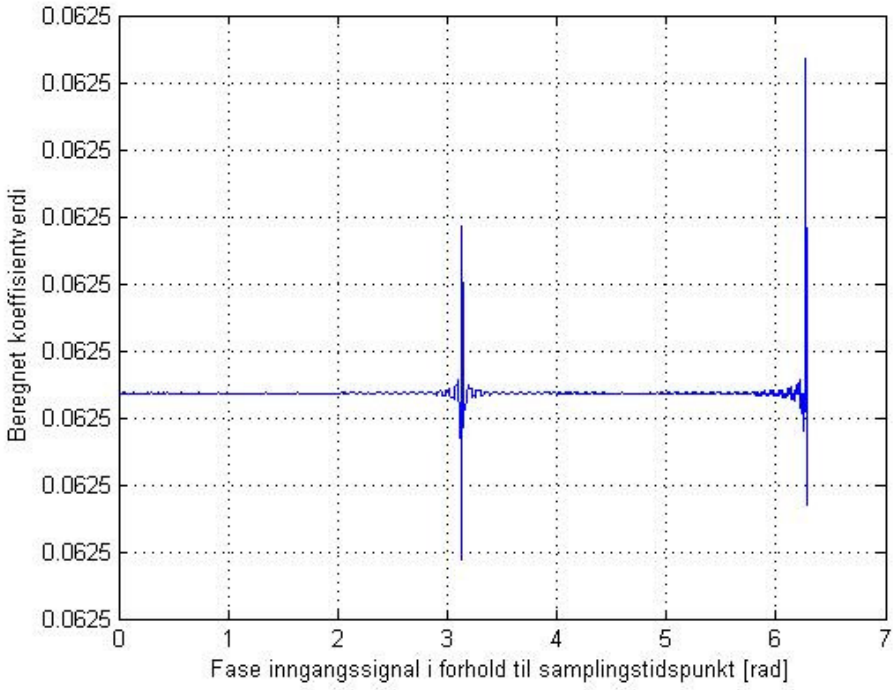
$$\alpha(t) = \frac{y_e(t) - y_l(t)}{\left(d_a + d_d\right) \cdot (1 - t^2)} \tag{46}$$

Med utgangspunkt i (46) er det nå mulig å beregne verdier for α ved å anta et kjent kontinuerlig inngangssignal.

**Beregning α som funksjon av periode på inngangssignal**

Vi kan først foreta en simulering der vi antar et kontinuerlig sinusformet inngangssignal på 1 KHz samplet med en frekvens $F_s$ = 44.1 kHz · 32 = 1411.2 KHz, som tilsvarer CD-lyd oversamplet med OSR=32.

Figur 34 viser de korrekte verdiene for α ved t=0 (se Figur 33), beregnet over en periode av det kontinuerlige inngangssignalet. Med denne simuleringen finner vi hvor mye den ideelle verdien for α vil variere med hvilke tidspunkter det kontinuerlige signalet samples i.

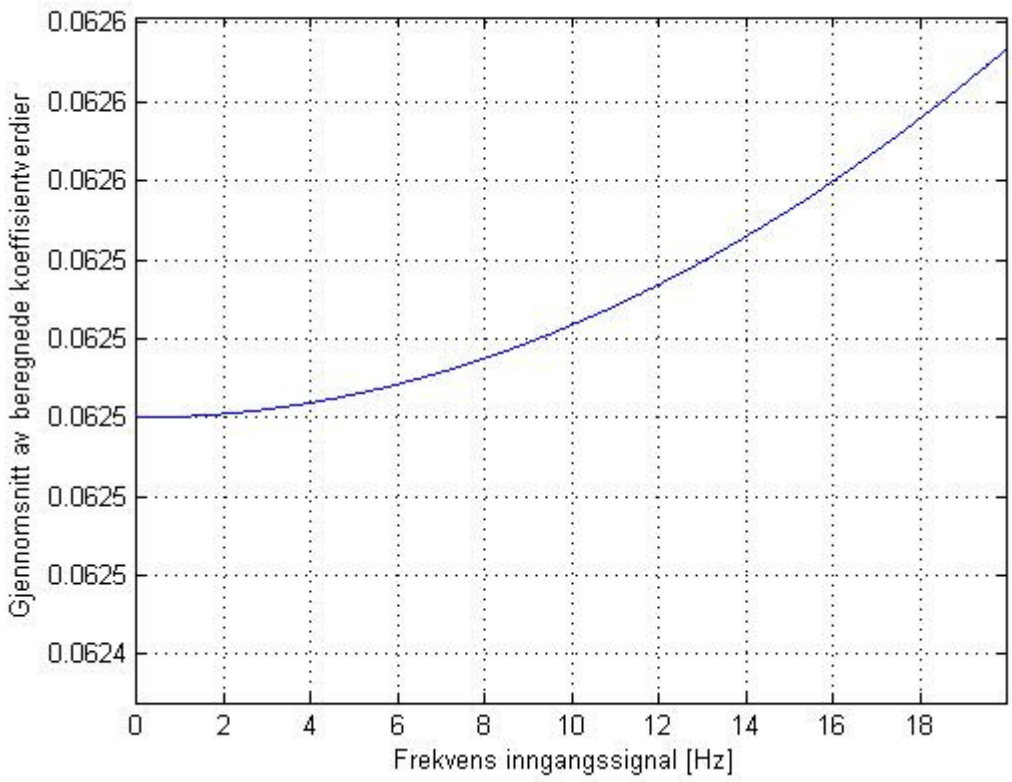



**Figur 34. Ideell α-verdi som funksjon av samplingstidspunkt, $\alpha_{max} - \alpha_{min} = 7.5 \cdot 10^{-9}$.**

Fra Figur 34 ser vi at den ideelle verdien for α varierer svært lite med sampletidspunktene.

**Beregning av α som funksjon av frekvens på inngangssignal**

Vi kan nå foreta en simulering for å finne sammenhengen mellom verdi for α og frekvens på inngangssignalet. Figur 35 viser gjennomsnittet av ideelle α-verdier over en signalperiode ved forskjellige signalfrekvenser.




**Figur 35. Ideell α-verdi som funksjon av signalfrekvens ved OSR=32 og $F_s$=44.1 kHz.**

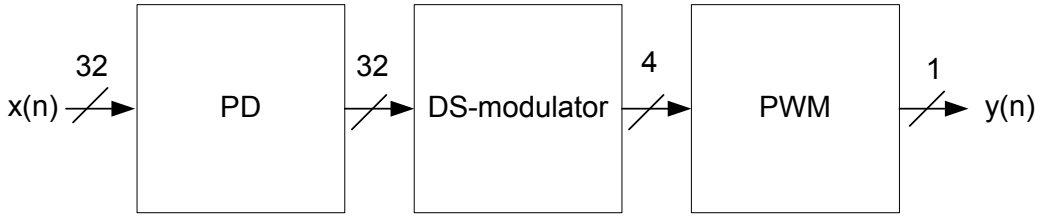
Vi ser at den ideelle verdien for α har moderat sammenheng med frekvensen på inngangssignalet, som hevdet i [10].

Simuleringene i figur Figur 34 og Figur 35 viser at en passende verdi for α åpenbart vil være **0.0625**.

Denne verdien er i tillegg særlig praktisk med tanke på hardware-implementering, da en multiplikasjon med $0.0625=2^{-4}$ bare vil innebære en enkel skift-operasjon.

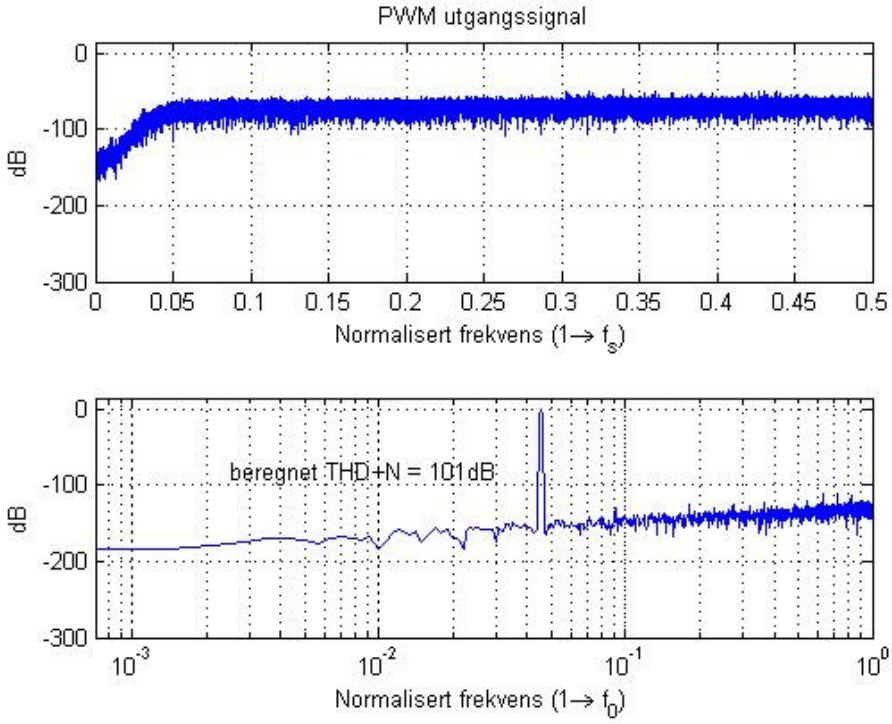
### 3.5.4 Simulering med predistortion

Simuleringen av den valgte algoritmen for predistortion er utført ved hjelp av oppsett i Figur 36, med en verdi for α lik 0.0625. Simuleringsoppsettet er identisk med oppsettet i Figur 30, bare med tillegg av predistortion-enheten. Disse to simuleringene kan derfor sammenliknes direkte for å finne forbedringen som følge av predistortion.



**Figur 36. Simuleringsoppsett for DS-modulator med PWM og *predistortion*.**

Figur 37 viser det simulerte utgangsspektrumet med den valgte predistortion-algoritmen. Sammenliknet med simuleringen i Figur 32 ser vi at de overharmoniske signalkomponentene er betydelig redusert. Predistortion gir en forbedring av THD+N fra -60 til -101 dB, som er tilfredsstillende i forhold til spesifikasjonene for DA-konverteren.




**Figur 37. PWM utgangssignal.**

Simuleringen i Figur 37 er for øvrig også foretatt med den mer avanserte predistortion-algoritmen (algoritme B) presentert i [10], uten at dette ga et bedre THD+N enn -100 dB. Dette kan skyldes at strukturen på DS-modulatoren i seg selv vil være med på å begrense hvor mye det ved hjelp av predistortion er mulig å forbedre THD, som blir nærmere omtalt i 3.6.

### 3.5.5 Enkelt- eller dobbeltsidig PWM?

Den prinsipielle forskjellen mellom enkeltsidig og dobbeltsidig PWM består i hvorvidt signalet moduleres ved å sammenlikne inngangssignalet mot et sagtann- eller trekantformet signal. Dette medfører at modulasjonsfrekvensen på utgangssignalet ved dobbeltsidig PWM vil reduseres til det halve i forhold til ved enkeltesidig, uten at informasjonsmengden i signalet reduseres.

Vi at det er ønskelig med så lav PWM-frekvens som mulig for å minimere effekten av de ikke-ideelle egenskapene på utgangen av FPGA-kretsen, og i så måte ville det være naturlig å velge dobbeltsidig PWM.

Dette imidlertid ikke like opplagt for systemet i denne oppgaven, siden signalet her blir modifisert ved predistortion samt kvantisert og støyformet før PWM-modualtoren. Det vil både i predistortion-algoritmen og DS-modulatoren måtte tas hensyn til hvilken PWM modulasjonstype som benyttes.

I DS-modulatoren er det for eksempel ikke tilstrekkelig å betrakte inngangssignalet til denne som en uniform samplet datasekvens (PCM), men heller som en sekvens av verdier som angir de naturlige transisjonstidspunktene for utgangssignalet, altså som pulsbredder. Denne problemstillingen kan bedre forstås fra [7].

På grunnlag av dette er det i dette prosjektet valgt å benytte enkeltsidig PWM da dette vil gi den enkleste implementeringen. Dette er for øvrig ikke opplagt, da vi egentlig ikke vet noe eksakt om sammenhengen mellom lydkvalitet og modulasjonsfrekvens.

## 3.6 Valg av modulatorstruktur

I 2.9 ble det gitt en gjennomgang av problemstillingene ved bruk av PWM, der vi blant annet så at det ikke er tilstrekkelig å bare betrakte inngangssignalet til PWM-modulatoren som en uniform samplet datasekvens (PCM), men heller som en sekvens av verdier som angir de naturlige transisjonstidspunktene for utgangssignalet, altså som pulsbredder.

Dette er det viktig å ta hensyn til også ved kvantiseringen og støyformingen av signalet i DS-modulatoren. Tradisjonelt har denne problemstillingen blitt oversett i denne sammenheng, og det naturlige PWM-signalet har derfor ofte feilaktig blitt sett på som et uniformt PCM-signal ved kvantisering og støyforming. Problemet med dette er at rett og slett at korreksjonen av kvantiseringen i DS-modulatoren ikke vil være naturlig, og da heller ikke gi de naturlige transisjonstidspunktene for det 1-bit utgangssignalet [7].

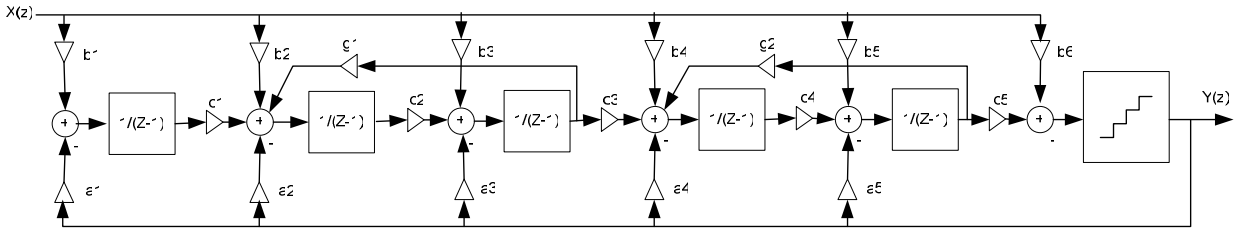
Simuleringer så langt viser at dette ikke er en like aktuell problemstilling i dette prosjektet, da det faktisk er mulig å oppnå de gitte spesifikasjonene selv om utgangssignalet betraktes som PCM, som tilfellet er ved simuleringene i 3.4.

Det er likevel jobbet en del med en moderne modulatorstruktur presentert i [7], kalt *Integral Noise Shaping*. Med utgangspunkt i denne strukturen og den mest avanserte av de to algoritmene for *predistortion* presentert i [10], er det gjort en del forsøk på å ytterligere forbedre THD+N utover det som ble oppnådd i 3.5.4, noe som ut fra spesifikasjonene for INS og nevnte predistortion-algoritme burde være innen rekkevidde. Det har imidlertid ikke lyktes å oppnå resultater med INS som favoriserer denne i forhold til mer tradisjonelle, og enklere, modulator-strukturer.

På bakgrunn av dette er det valgt en standard 5. ordens *cascade of resonators, feedback form* (CIFB) struktur som vist i Figur 38. Denne strukturen tilbyr stor frihet i forhold til modulatorens overføringsfunksjon gjennom koeffisientsettene A, B, G og C. Eksempelvis gjør tilbakekoplingene gjennom g1 og g2 det mulig å velge nullpunktplasseringene for $N_{TF}$, noe som er ønskelig i dette prosjektet.

Ved hjelp av *The Delta-Sigma Toolbox 7.1* [31] for Matlab er det generert koeffisienter på bakgrunn av den valgte $N_{TF}$. Her viser det seg at *feed-forward* koeffisientene C, og koeffisient $b_5$ vil være lik 1, og multiplikasjonene med disse kan derfor utelukkes ved implementering av DS-modulatoren.

Det er også foretatt simuleringer med kvantiserte koeffisienter for å finne nødvendig oppløsning for hardware-implementering. Det ser ut til at det oppnås tilstekkelig stabilitet og ytelse ved 18-bits representasjon av koeffisientene. Denne nøyaktigheten er delvis valgt på bakgrunn av at FPGA-kretsen systemet skal implementeres i inneholder 18-bits multiplikatorer[13].




**Figur 38. CIFB modulatorstruktur.**

## 3.7 Interpolering og interpoleringsfiltre

### 3.7.1 Generelle krav

Den nødvendige interpoleringsaktoren I er gitt av den ønskede oversampling-raten (OSR), som ved dimensjonering av DS-modulatoren ble fastsatt til 32.

Når det gjelder kravet til interpoleringsfiltrene, er ikke spesifikasjonene for disse like opplagt ettersom frekvenskomponentene som oppstår ved interpolering uansett vil falle utenfor det hørbare området.

Hovedgrunnene til at det likevel er hensiktsmessig å benytte interpoleringsfiltre er som følger [1]:

- DS-modulatoren må behandle mindre total signalenergi, som medfører lavere krav til dynamisk område og bedre stabilitet i modulatoren.
- Oppgaven til det analoge filteret på utgangen vil bli lettere da det må filtreres bort mindre støy utenfor basisbåndet.

Passbånd-rippel bør ikke være hørbar. I [16] er det studert en rekke eksisterende høykvalitets DA-konvertere. For disse varierer denne parameteren fra 0.001 dB til 0.0001 dB. Dette kan også være et utgangspunkt for filtrene i dette prosjektet. Vi kan velge den mest moderate av disse verdiene. Altså:

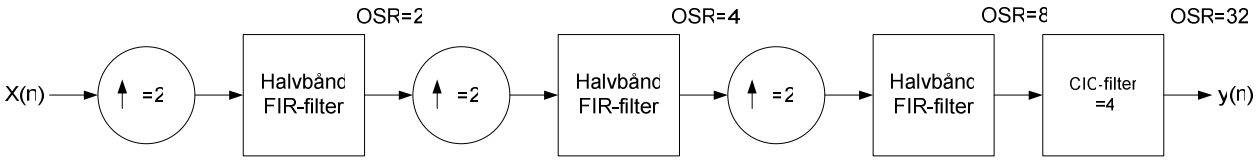
- Passbånd-rippel bør ikke overstige 0.001 dB.

Vi vet fra før at utgangssignalet for en DS-modulator vil inneholde støyeffekt utenfor basisbåndet. Et naturlig mål for interpoleringsfiltrenes demping i stoppbåndet vil være å dempe de uønskede frekvenskomponentene fra interpoleringen til et lavere enn støygulvet fra DS-modulatoren. Vi kan fastsette følgende krav som et utgangspunkt:

- Demping i stoppbånd børe være tilstrekkelig til å dempe aliasfrekvenser til et nivå 10-15 dB lavere enn støygulvet fra DS-modulatoren.

### 3.7.2 Struktur

Figur 39 viser den valgte strukturen for interpolatoren med filtre.




**Figur 39. Valgt struktur for interpolering og filtrering.**

Interpolering og filtrering blir ofte realisert av flere interpolatorer/filtre i kaskade. Det viser seg at flere interpolatorer og filtre i kaskade vil gi et færre antall regneoperasjoner totalt enn det man ville behøve med bare et enkelt filter [3]. Dette har sammenheng med at kravet til interpoleringsfiltrenes transisjonsbånd vil avta utover i kjeden etter hvert som signalet blir mer og mer oversamplet.

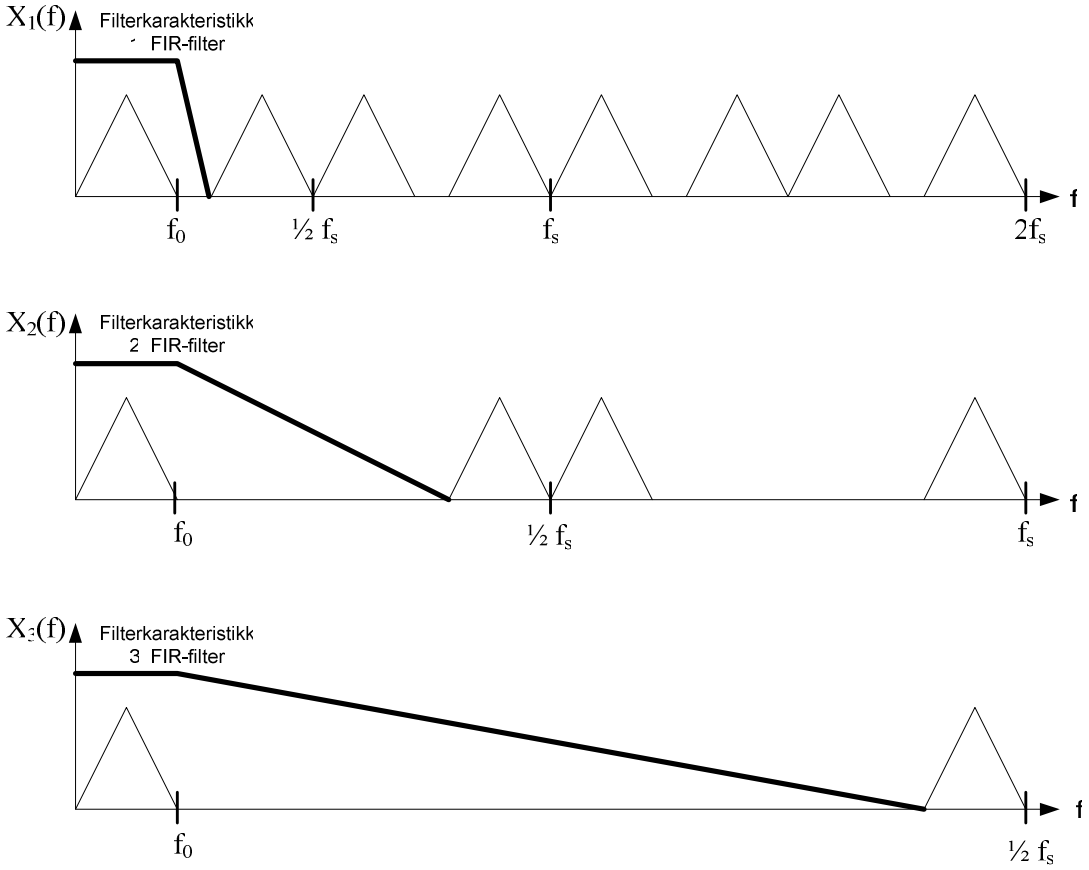
Det er særlig gunstig å foreta en interpolering med I=2 i hvert trinn, da dette gjør det mulig å benytte halvbånd FIR-filter som beskrevet i 2.7.4. Vi vet at i en effektiv realiseringer av et slikt filter er tilnærmet annenhver filterkoeffisient lik 0. Når i tillegg annenhver sample i det interpolerte signalet er lik 0 etter ”zero-stuffing”, vil i størrelsesorden ¾ av alle multiplikasjonene i interpoleringsfilteret bli lik 0. Dette åpner for en effektiv implementering i hardware.

Det er utelukkende benyttet FIR- og CIC-filter da dette gjør det mulig å oppnå lineær-fase, en egenskap som er ønskelig i forbindelse med audio [22].

### 3.7.3 Dimensjonering av halvbånd FIR-filter

I Figur 12 så vi hvordan det nødvendige transisjonsbåndet for et interpoleringsfilter er gitt av størrelsen på signalets basisbånd i forhold til samplefrekvensen før interpolering.

Figur 41 viser hvordan størrelsen på transisjonsbåndene for FIR-filtrene i dette systemet vil øke for hvert trinn i interpoleringskjeden, der $X_1$-$X_3$ tilsvarer frekvensspekteret på inngangen til hvert av FIR-filtrene.

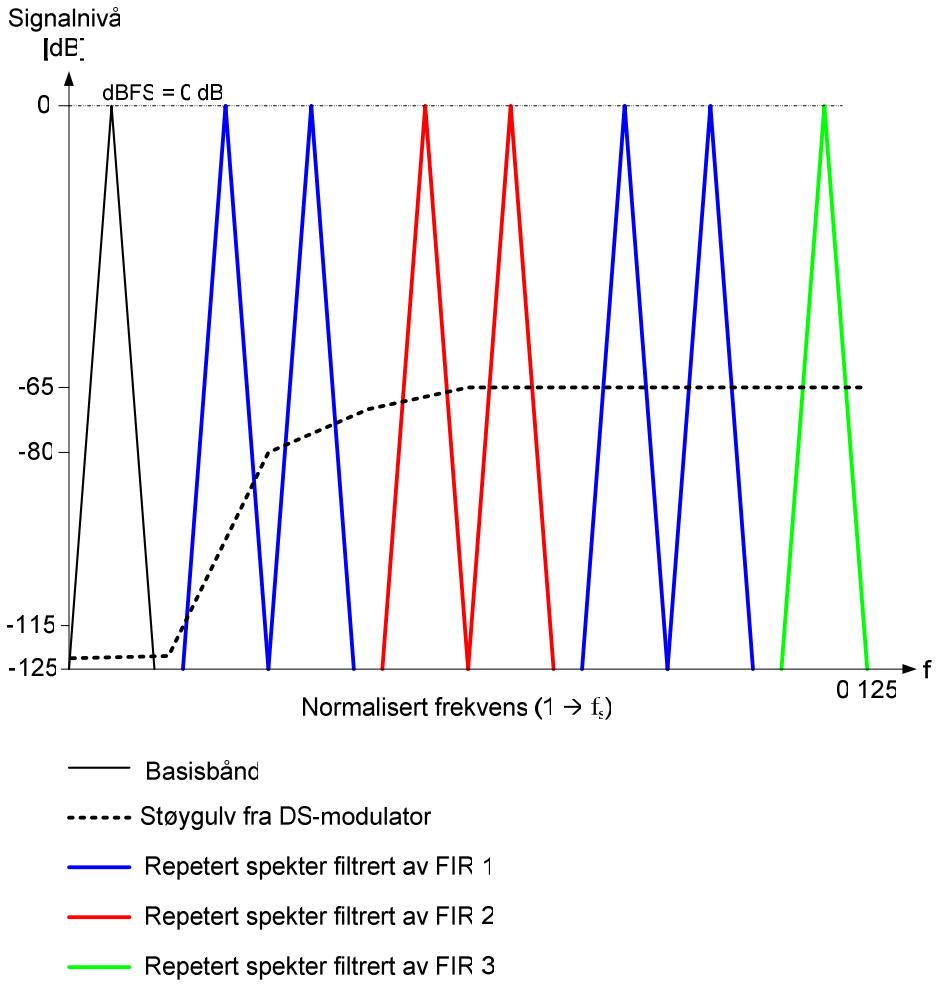



**Figur 40. Inngangssignal på inngang av FIR interpoleringsfiltre.**

Med utgangspunkt i Figur 40 og spesifikasjonene for CD-lyd [33] kan vi finne de nødvendige verdiene for filtrenes transisjonsbånd:

- FIR-filter 1: Transisjonsbånd 20 - 24.1 kHz ved $F_s$=88.2 kHz.
- FIR-filter 2: Transisjonsbånd 20 - 68.2 kHz ved $F_a$=176.4 kHz.
- FIR-filter 3: Transisjonsbånd 20 - 156.4 kHz ved $F_s$=352.8 kHz.

I simuleringen i Figur 37 så vi formen på det forventede støygulvet fra DS-modulatotren mens Figur 40 viser hvilke frekvensområder som vil måtte filtreres av de forskjellige FIR-filtrene i kjeden. Disse to figurene kan derfor sammenfattes og brukes som utgangspunkt for å fastslå nødvendig stoppbånddemping for hvert av de 3 FIR-filtrene. Dette er illustrert i Figur 41.




**Figur 41. Grunnlag for valg av spesifikasjoner for FIR-filter.**

Kravene til stoppbånddempingen kan som nevnt tidligere fremstå noe diffuse da dette båndet uansett er utenfor det hørbare området. Hvis vi tar sikte på å dempe de repeterte frekvensene 10-15 dB under støygulvet fra DS-modultatoren, kan vi fastsette følgende krav til stoppbånddempingen i FIR-filtrene:

- FIR-filter 1: Stoppbånddemping = 100 dB.
- FIR-filter 2: Stoppbånddemping = 90 dB.
- FIR-filter 3: Stoppbånddemping = 80 dB.

### 3.7.4 Realisering av FIR-filter i Matlab

De 3 FIR-filteren er realisert ved hjelp av *filter design & analysis tool* i Matlab. Dette er et verktøy som kan generere koeffisienter for digitale filtre ut fra gitte filterspesifikasjoner.

Med tanke på implementering i hardware er det ønskelig å oppnå filterspesifikasjonene med et færrest mulig antall filterkoeffisienter, hvor hver koeffisient representeres med et lavest mulig antall bit (nøyaktighet). Et stort antall koeffisienter vil både kreve mye lagringsplass i ROM

og et stort antall beregninger, mens høy nøyaktighet i tillegg vil kreve komplekse multiplikatorer og adderere. Redusert nøyaktighet, eller kvantisering av koeffisientene, vil igjen redusere filterets ytelse.

Alle de 3 filtrene er realisert som halvbåndfilter ved hjelp av teknikken beskrevet i 2.7.4.

Tabell 2 viser dataene for den endelige realiseringen av de 3 filtrene som er funnet mest gunstig med tanke på punktene beskrevet ovenfor.

| Filter | RMS stoppbånd-demping [dB] | Minium stoppbånd-demping [dB] | Passbånd rippel [dB] | # koeffisienter ≠ 0 | #bit nøyaktighet per koeffisient |
|---|---|---|---|---|---|
| **1** | 101.8 | 93 | 0.0004 | 2 x 32 | 17 |
| **2** | 90.5 | 84.3 | 0.0008 | 2 x 8 | 14 |
| **3** | 80.1 | 74.3 | 0.0015 | 2 x 4 | 9 |

**Tabell 2. Parametere for FIR-filtere.**

### 3.7.5 Dimensjonering av CIC-filter

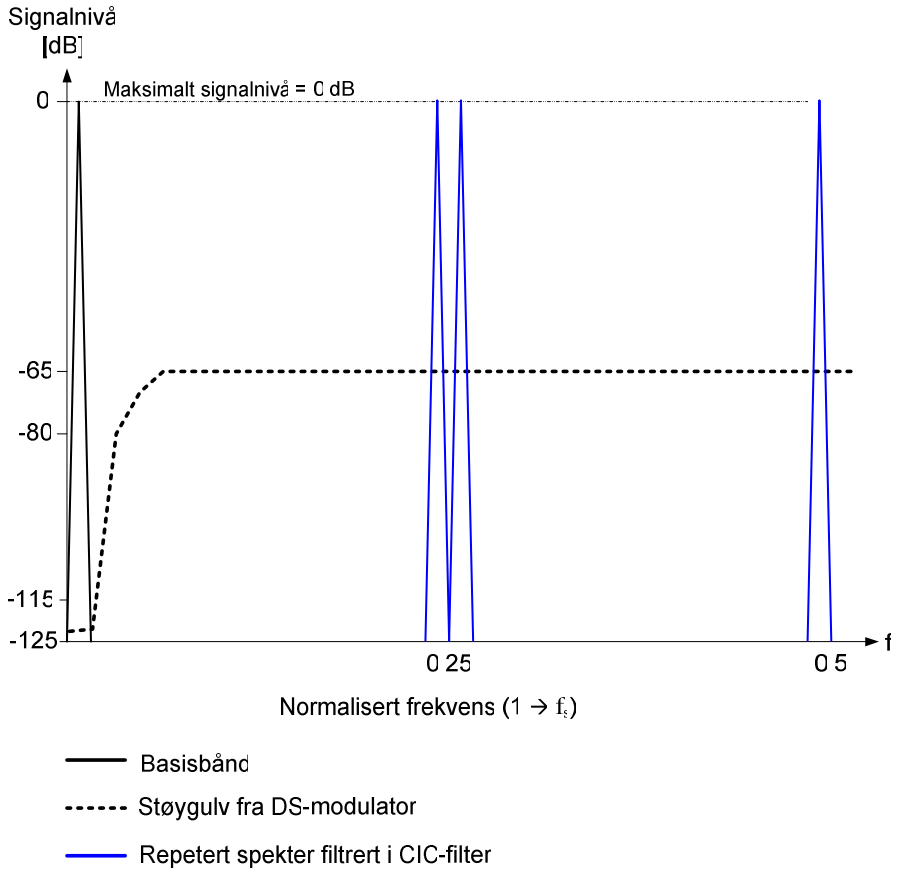
Det siste leddet i interpoleringskjeden, CIC-filteret, vil både interpolere signalet med faktoren I og filtrere bort aliasfrekvensene som følger av interpoleringen. Oversampling-raten (OSR) for systemet er satt til 32, og med en interpolering på $2^3$=8 i de foregående trinnene, vil CIC-filteret måtte interpolere signalet med en faktor I=32/8=4.

Vi kan starte med å repetere frekvensresponsen for dette filteret som ble gitt i 2.7.5:

$$|H(f)| = \left| \frac{\sin(\pi M f)}{\sin\left(\pi \frac{M f}{I}\right)} \right|^N$$

I uttrykket for filterets frekvensrespons er f normalisert til 1 i forhold til Nyqiust-raten før interpolering, det vil si at uttrykket må evalueres for f=0-2 for å finne frekvensresponsen i hele Nyquist-båndet etter interpolering med faktoren 4. Frekvensresponsen for et CIC-filter vil ikke ha en typisk lavpass-karakteristikk, der man har et klart definert pass- og stoppbånd, men i stedet ha topp- og bunnpunkter distribuert over hele båndet. Et CIC-filteret må derfor dimensjoners slik at disse nullpunktene faller i de områdene av spektret der aliasfrekvensene fra interpoleringen oppstår.

Vi kan finne den nødvendige plasseringen for bunnpunktene ved å utvide Figur 41 til å gjelde hele Nyquist-båndet etter interpoleringen. Dette er vist i Figur 42, der bare frekvenskomponentene som må filtreres av CIC-filteret er tatt med.




**Figur 42. Grunnlag for valg av spesifikasjoner for CIC-filter.**

Alle frekvenskomponentene som må filtreres av CIC-filteret ligger altså i områder av frekvensbåndet der kvantiseringsstøyen fra DS-modulatoren er høyest. Dempingen i stoppbåndene (rundt nullpunktene) kan derfor settes lik kravet for det 3. FIR-filteret, altså til 80 dB.

Siden OSR her er gitt, ser vi fra overføringsfunksjonen i (38) at vi bare har frihet til å variere N og M. Evaluering av overføringsfunksjonen viser følgende sammenhenger med tanke på disse parametrene:

- Økning av N øker demping i stoppbånd, men øker også passbånd dropp
- Nullpunkter vil falle på frekvenser n 1/M referert til Nyquist-raten før interpolering som tilsvarer n 1/(M·I) referert til sampleraten $f_s$ etter interpolering.

Vi kan starte med å finne en passende verdi for parameteren M. Fra Figur 42 ser vi at det er ønskelig med nullpunkter i 0.25 og 0.5 normaliser til $f_s$. Dette oppnås med **M=1**.

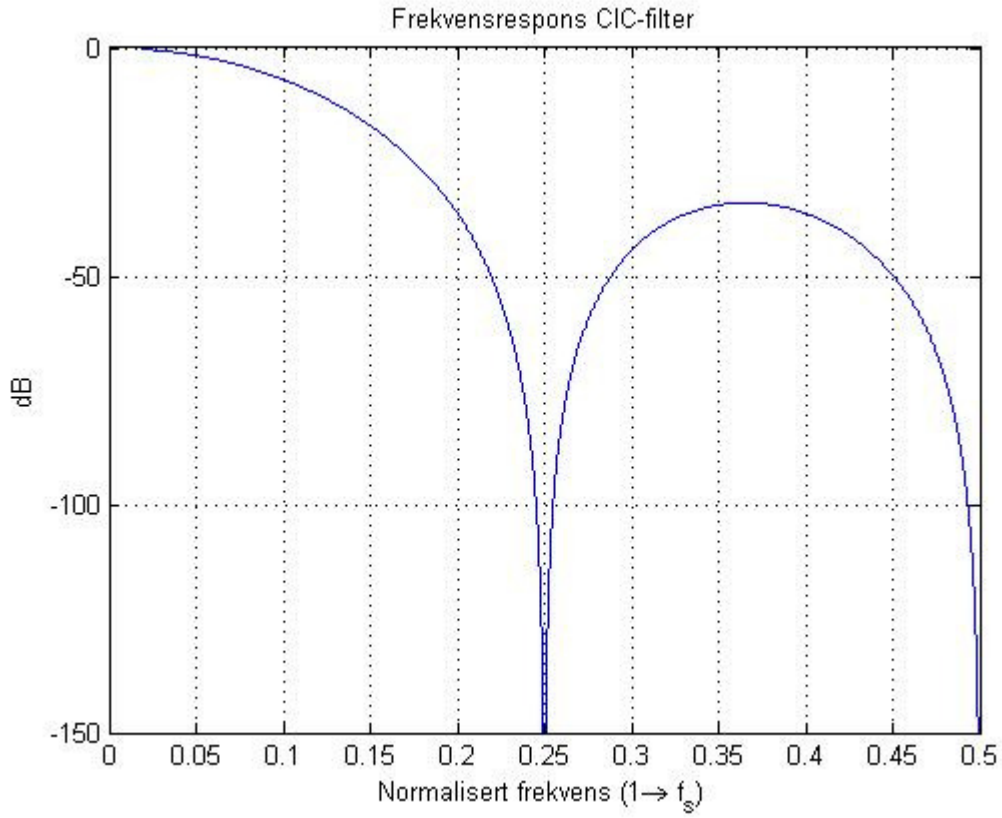
Videre simuleringer med M=1 viser at et passende antall trinn er **N=3**.

De Endelig dataene for CIC-filteret med de valgte parameterne er vist i Tabell 3 mens Figur 43 viser frekvensresponsen for filteret korrigert for filterforsterkningen. Vi ser at filteret overholder de ønskede spesifikasjonene med unntak av passbånd dropp (kan sammenliknes med rippel) som er på 0.1 dB. Det er likevel valgt å gå videre med dette filteroppsettet da de valgte parameterverdiene uansett synes å være de mest hensiktsmessige. Eksempelvis vil N=3

forbedre passbånd dropp til 0.09 dB, men til gjengjeld redusere stoppbånddempingen til 54 dB.

| Filter | RMS demping 1. stoppbånd [dB] | Minimum demping 1. stoppbånd [dB] | RMS demping 2. stoppbånd [dB] | Minimum demping 2. stoppbånd [dB] | RMS demping 1. og 2. stoppbånd sammenslått [dB] | Passbånd dropp [dB] |
|---|---|---|---|---|---|---|
| **CIC** | 77.9 | 68.3 | 87 | 78.7 | 80 | 0.1 |

**Tabell 3. Parametere for CIC-filter.**




**Figur 43. Frekvensrespons for CIC-filter med parametere M=1 og N=3.**

## 3.8 Oppsummering av design

Oppsettettet ved simulering i 3.4.8 synes altså å være hensiktsmessig med tanke på spesifikasjonene for DA-konverteren, og dette oppsettet vil derfor være utgangspunktet for den implementeringen av DS-modulatoren.

Dette oppsettet har en OSR=32 som tillater 4-bitskvantisering, eller 16-nivåers pulsbredde-modulasjon. Det er fortatt valg av algoritme for predistortion, og simuleringer viser at det skal være mulig å oppnå et THD+N på -101 dB, 0 dBFS.

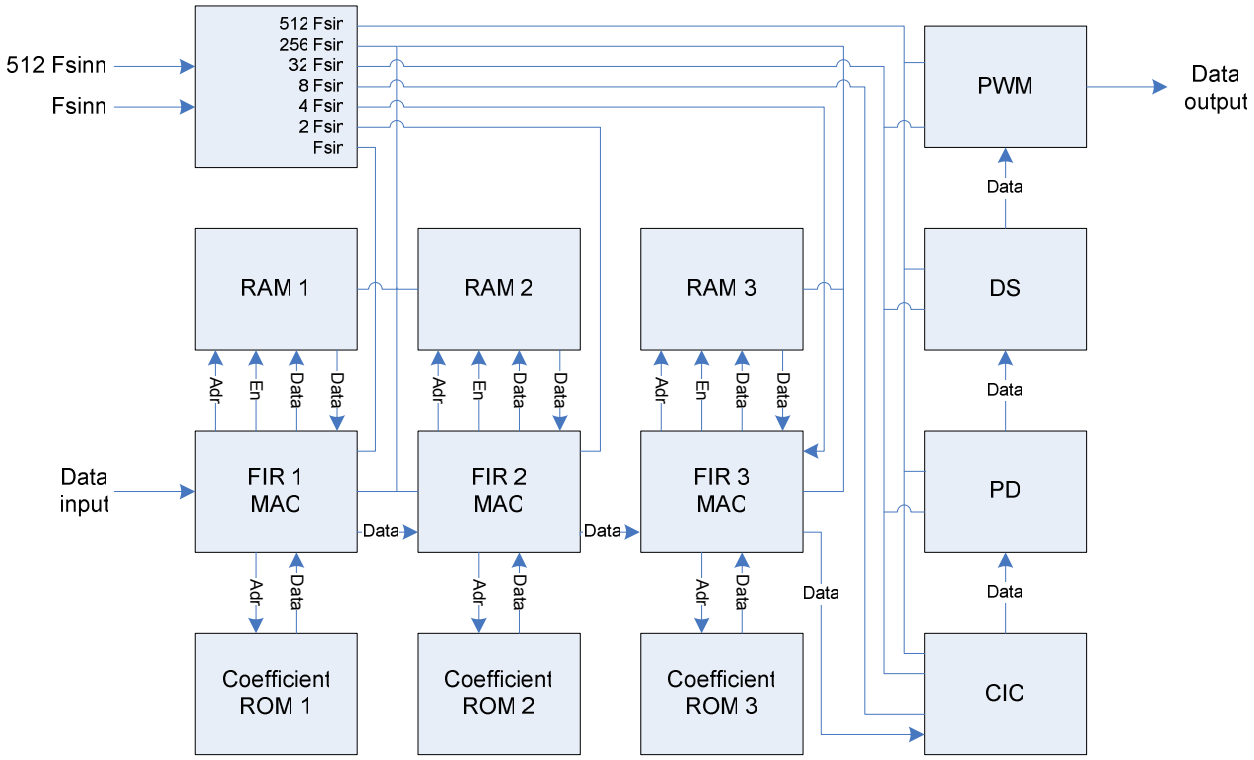
# 4 Implementering

*Det vil i dette kapittelet bli gitt en gjennomgang av implementeringen av systemet i FPGA. Figur 44 viser topologien for Verilog-implementeringen av DA-konverteren. Denne er oppbygd på samme modulære måte som utgangspunktet for designet vist i Figur 25. Fordelen med en slik modulær oppbygning er at hver modul kan testes og verifiseres separat, før de settes sammen i det endelige systemet. Dette gjør det betraktelig enklere å lage en velfungerende og fleksibelt implementering. På den andre side begrenser dette muligheten for deling av hardware-ressurser mellom modulene, noe som vil kunne gi en ikke areal-effektiv løsning.*

*Ved implementering av modulene er det lagt vekt på deling av hardware-ressurser innenfor hver modul, da en av de største utfordringene har vært å møte disse begrensningene gitt av FPGA-kretsen systemet skal implementeres på. Det er i systemet tilgang på en klokke med frekvens 16 $f_s$ (der $f_s$ er samplerate ut av kretsen), noe som innebærer at funksjonelle enheter kan allokeres, eller gjenbrukes, over 16 trinn for hver utgangsverdi som skal beregnes. I gjennomgangen nedenfor vil bare topologien for hver enkelt modul vist, mens det i appendiks er vedlagt full verilog-kode for disse.*

*Ved implementeringen har det vært lagt vekt på fleksibilitet ved å gjøre systemet mest mulig paramteriserbart, slik at ytelse og bruk av hardware-ressurser sener skal kunne justeres etter behov.*

*Det er ikke fokusert særlig på minimering av strømforbruk, både fordi det ikke ble formulert noen krav til dette, og fordi fokus for oppgaven har vært på DA-konverterens funksjonalitet.*




**Figur 44. Toppnivå topologi for *hardware*-implementering.**

## 4.1 Halvbånd FIR og interpolering med I=2

For FIR-filterene i kretsen er det benyttet en såkalt *multiply-accumulate* (MAC) struktur. Dette innebærer at implementeringen består av 1 multiplikator og 1 akkumulator, hvor hver utgangsverdi beregnes ved å akkumulere utgangsverdier fra multiplikatoren, der inngangsverdiene til multiplikatoren er koeffisientverdier fra ROM og data fra et inngangsbuffer.

Implementeringen av FIR-filteret foretar også interpoleringen med faktoren I=2. Dette vil si at det beregnes to utgangsverdier fra filteret for hver inngangsverdi. Vi vil nå se på sammenhengen mellom inngangsverdiene og utgangsverdiene for denne modulen, der det utnyttes at inngangssignalet "zero-stuffes" (interpolers) og at FIR-filteret har symmetriske koeffisienter der annenhver koeffisient har verdi 0.

Vi kan først tenke oss inngangssignalet i filterets inngangsbuffer følgende måte:

$$d_{buffer}(n) = \{s_{-(2N-1)} \;\; s_{-(2N-2)} \;\; \ldots \; s_{-N} \; \ldots \;\; s_{-1} \;\; s_0\}$$
$$\uparrow$$

Inngangssignalet i $S_{buffer}(n)$ interpolert med I=2 vil da tilsvare:

$$d_{interpolerct}(n) = \{\ s_{-(2N-1)}\ \ 0\ \ s_{-(2N-2)}\ \ 0\ \ \ldots\ \ s_{-N}\ \ldots\ \ s_{-1}\ \ 0\ \ s_0\ \ 0\ \}$$
$$\uparrow$$

Videre kan vi anta et sett symmetriske filterkoeffisienter på formen:

$$h(n) = \{\ c_0,\ 0,\ c_1,\ 0,\ c_2,\ \ldots,\ c_{N-1},\ 1,\ c_{N-1},\ 0,\ \ldots,\ c_2,\ 0,\ c_1,\ 0,\ \ c_0\ \}$$
$$\uparrow$$

der koeffisientverdiene ≠ 0 eller 1 lagret i ROM vil tilsvare:

$$c_{rom}(n) = \{c_0,\ c_1,\ \ c_2,\ \ldots,\ \ c_{N-1}\ \}$$
$$\uparrow$$

Med utgangspunkt i signalene og filterkoeffisientene ovenfor kan vi nå finne de to utgangsverdiene y(n) og y(n+1) fra FIR-filteret:

$$y(n) = S_{interpolert}(n) * h(n)$$

$$= c_0 s_0 + 0 + c_1 s_{-1} + 0\ c_2 s_{-2} + \ldots + c_{N-1} s_{-(N-1)} + c_{N-1} s_{-N} + 0 + c_{N-2} s_{-(2N+1)} + 0 + c_0 s_{-(2N-1)}$$

$$= c_0 s_0 + c_1 s_{-1} + c_2 s_{-2} + \ldots + c_{N-1} s_{-(N-1)} + c_{N-1} s_{-N} + c_{N-2} s_{-(2N+1)} + \ldots + c_0 s_{-(2N-1)}$$

$$= \sum_{k=0}^{N-1} d_{buffer}(-k) \cdot c_{rom}(k) + \sum_{i=N}^{2N-1} d_{buffer}(-i) \cdot c_{rom}(2N-1-i)$$

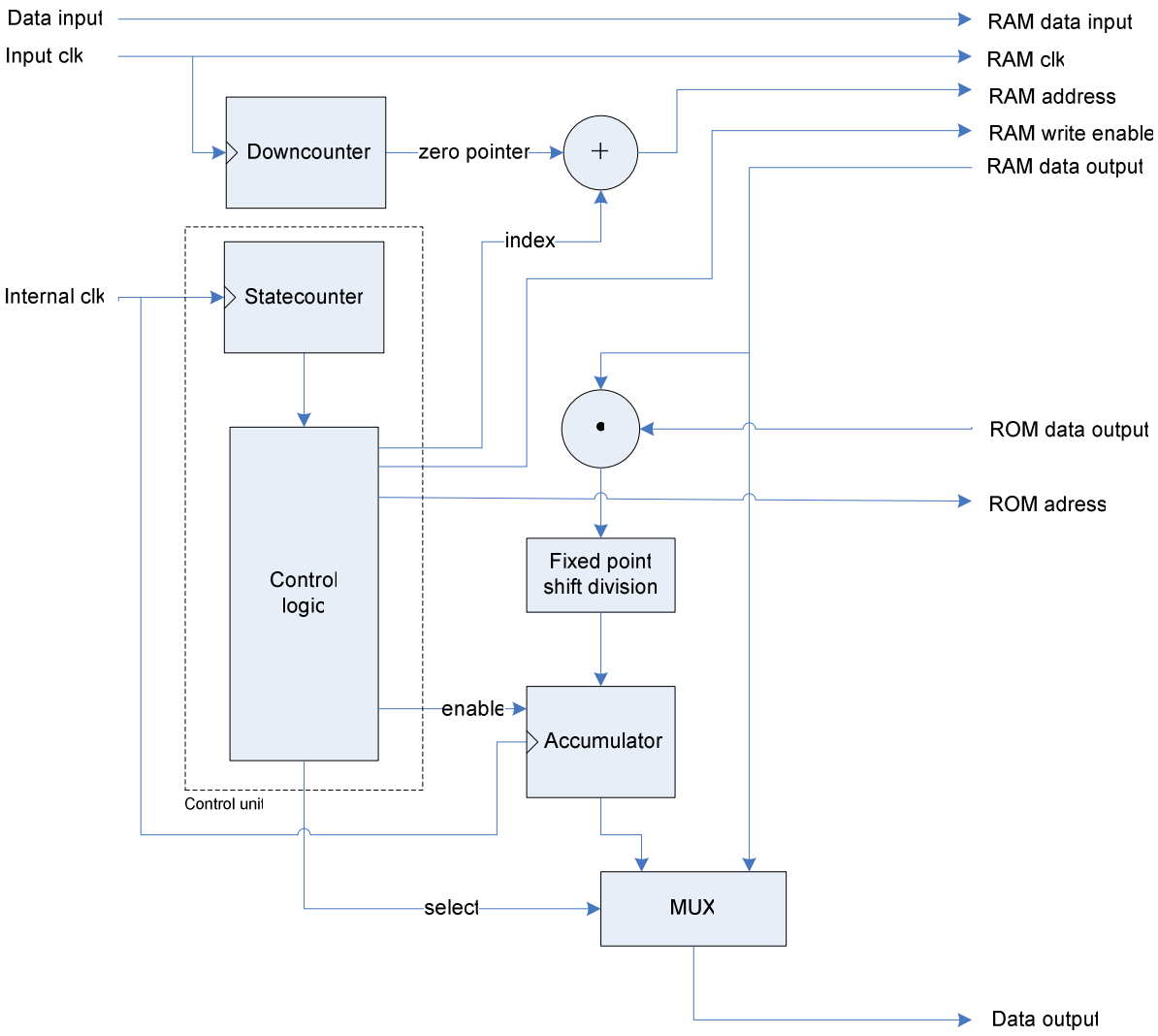
$$y(n+1) = S_{interpolert}(n+1) * h(n+1)$$
$$= 1 \cdot s_{-N}$$
$$= S_{buffer}(-N)$$

Altså tilsvarer annenhver utgangsverdi fra filteret en verdi dirkete fra inngangsbufferet.

Figur 45 viser prinsippet for hardware-implementeringen av FIR-filtrene med interpolatorer. Denne strukturen brukes for alle de 3 FIR interpolatorene.

For inngangsbufferene benyttes det RAM som finnes i FPGA-kretsen systemet skal programmers i. Det benyttes sirkulær buffering, det vil si at en sirkulerende adressepeker bestemmer hvor nye inngangsverdier skal legges i RAM, i stedet for at verdien skiftes inn, noe som ville medfører at alle verdiene i inngangsbufferet (RAM) måtte skifte plassering for hver nye inngangsverdi.

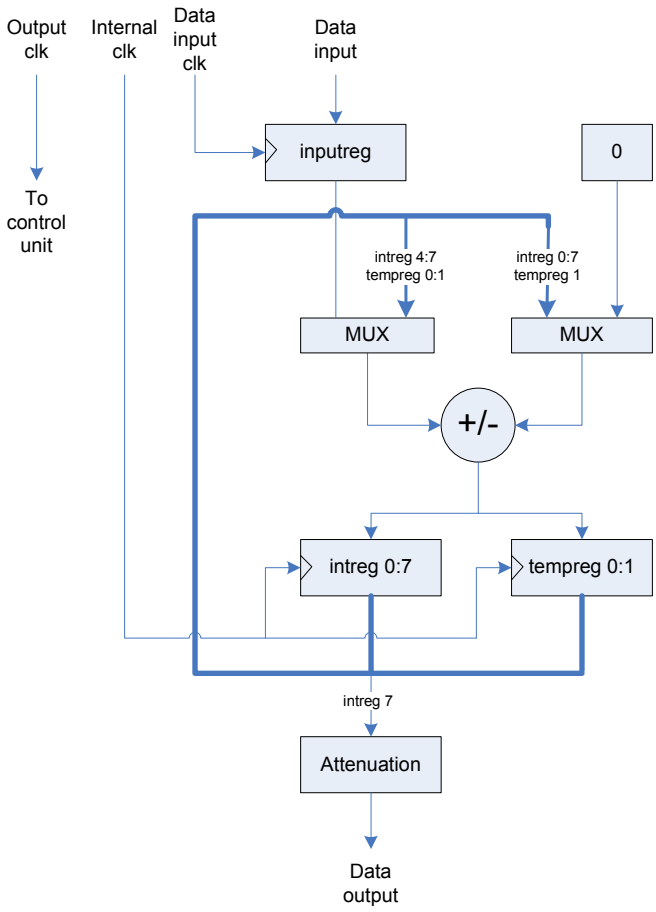
Det for øvrig også laget en utgave av denne modulen om benytter registre i stedet for RAM for buffering. Denne er ikke hensiktsmessig å benytte ved implementering i FPGA, men kan være interessant for et eventuelt ASIC-design.




**Figur 45. Topologi for *hardware*-implementering av FIR-filter.**

## 4.2 CIC-filter

Figur 46 viser prinsippet for hardware-implementeringen av CIC-filteret.
I figuren er for enkelthets skyld kontrollenheten utelukket. CIC-filteret betår i prinsippet av 6 akkumulatorer, som her blir realisert ved hjelp av en enkelt adderer og 9 registere.




**Figur 46. Topologi for *hardware*-implementering av CIC-filter.**

## 4.3 Predistortion

Figur 47 viser prinsippet for hardware-implementeringen av predistortion-modulen.
I figuren er for enkelthets skyld kontrollenheten utelukket.

Fra pseudo-koden i 3.5.2 for den valgte predistortion-algortimen ser vi at denne krever én divisjon samt en rekke multiplikasjoner. Ved hardware-implementering er særlig en divisjon svært ressurskrevende. Det har derfor vært hensiktsmessig å ved hjelp av rekkeutvikling erstatte denne divisjonen med flere multiplikasjoner.

Algoritmen inneholder en divisjon på formen:

$$A = \frac{B}{1 - C}$$

der C er gitt som ½ $s_2$-$s_1$.

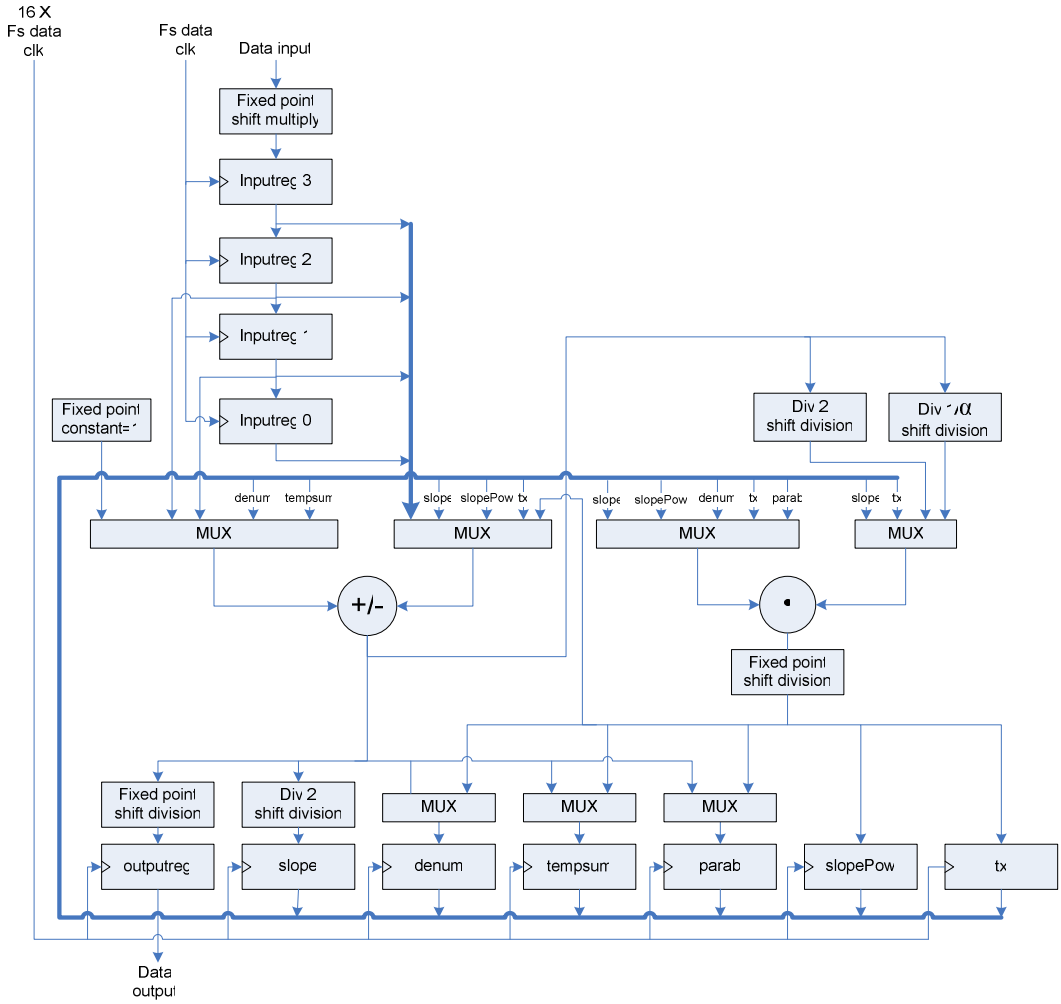
Det er rett fram å se at C vil variere rundt 0, og det er derfor naturlig å benytte Taylor rekkeutvikling av A med tanke på C som gir:

$$A \approx \sum_{k=0}^{N} B \cdot C^{k}$$

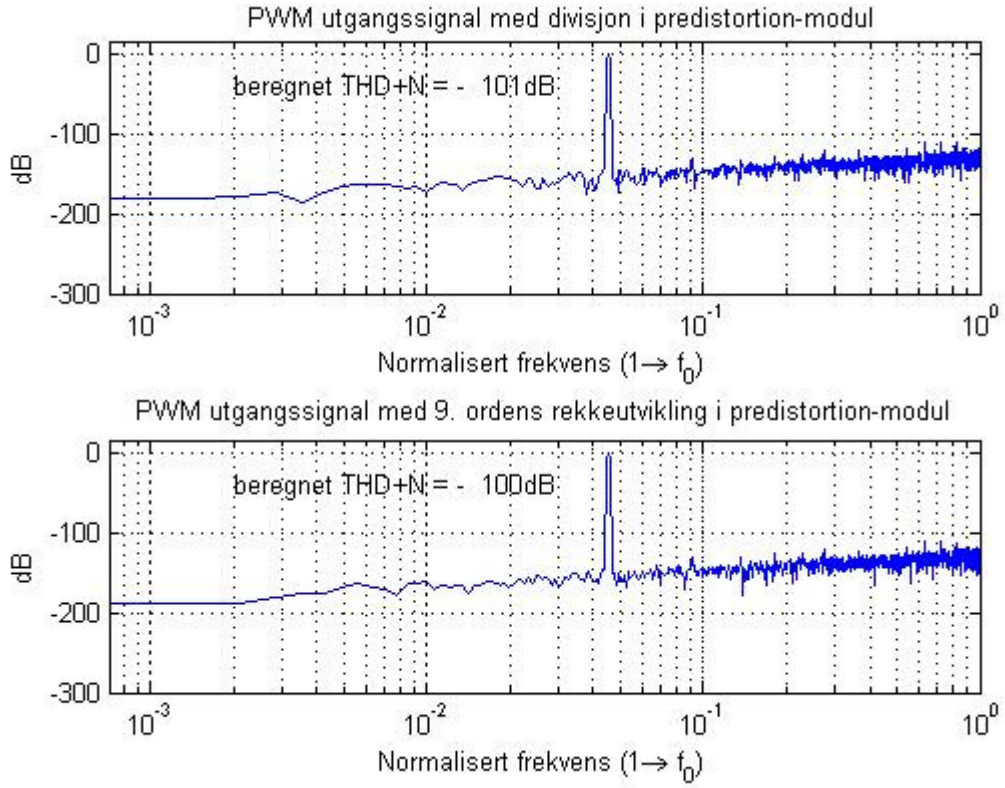
Der N angir ordenen på rekkeutviklingen.

Predistortion-algortimen inneholder i realiteten bare 3 multiplikasjoner som faktisk behøver en multiplikator i hardware, mens de resterende multiplikasjonene kan realiseres som enkle skift-operasjoner. Dette gir mulighet for høyere ordens rekkeutvikling da multiplikatoren ville kunne benyttes til dette i 13 av de 16 mulige operasjonstrinnene.

I denne implementeringen er imidlertid dette ikke utnyttet til fulle, hvor det er valgt å benytte multiplikatoren i 8 trinn, noe som gir en 9. ordens rekkeutvikling. Figur 48 viser en simulering der utgangssignalet ved divisjon i predistortion-modulen sammenliknes med utgangssignalet der denne divisjonen er erstattet med en 9. ordens rekkeutvikling. Vi ser at dette gir tilfredsstillende resultat.



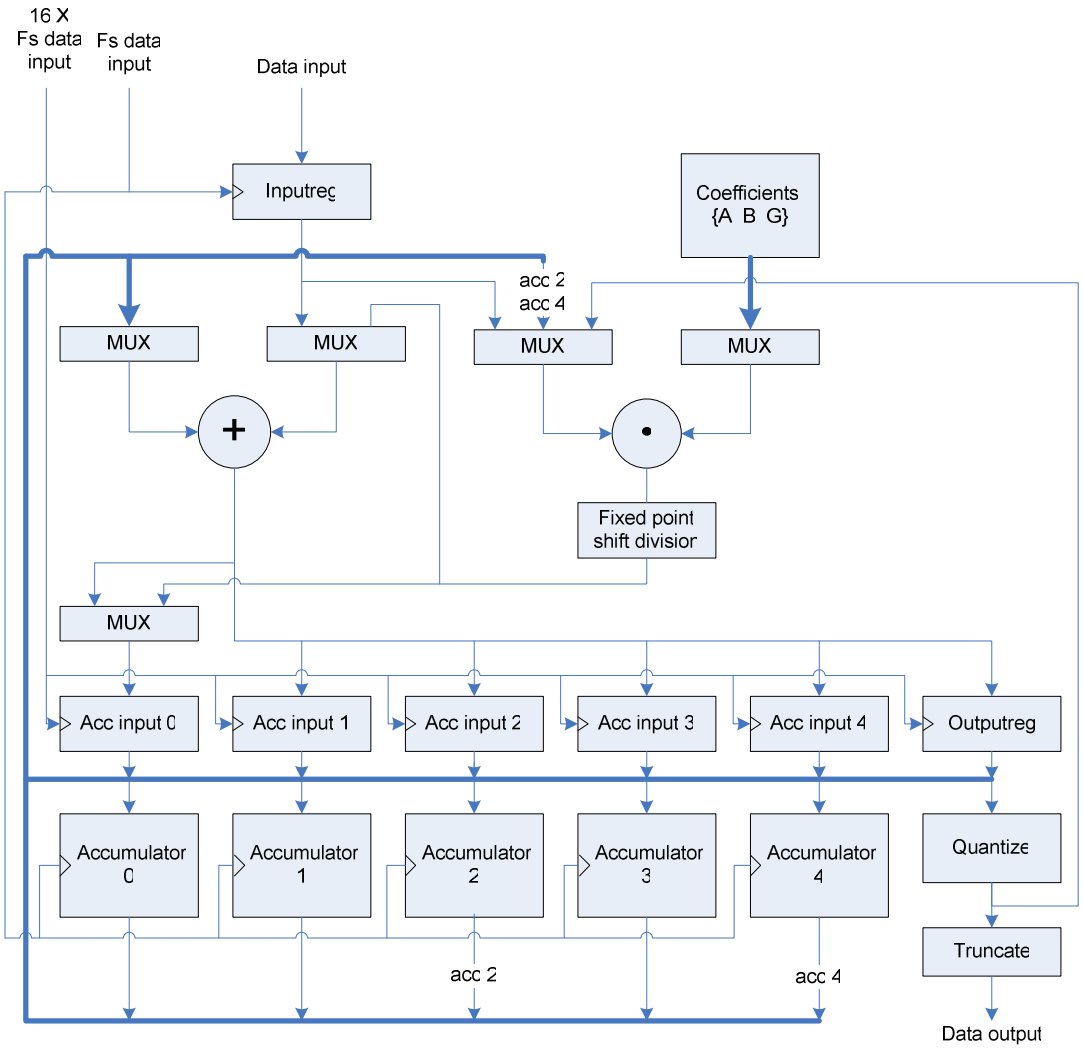

**Figur 47. Topologi for hardware-implementering av predistortion-enhet.**




**Figur 48. Sammenlikning av utgangssignal ved divisjon i predistortion mot utgangssignal ved rekkeutvikling i predistortion.**

## *4.4 DS-modulator*

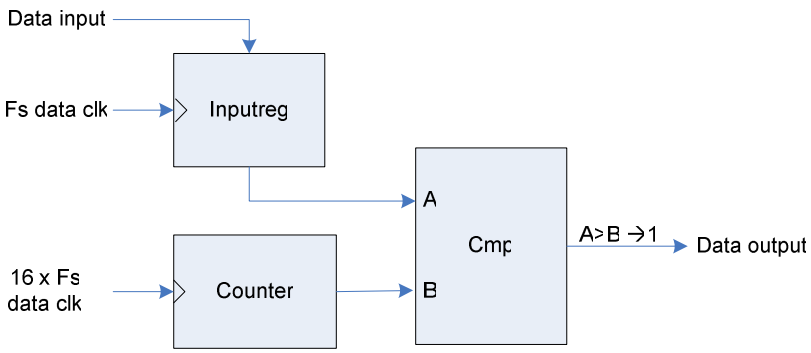
Figur 49 viser prinsippet for hardware-implementeringen av delta-sigma-modulatoren. I figuren er for enkelthets skyld kontrollenheten utelukket. For DS-modulatoren er det benyttet en 5. ordens CIFB-strukttur som vist i Figur 38. Denne strukturen inneholder 5 akkumulatorer, 12 addisjoner og 12 multiplikasjoner (ved koeffisientsett C og $b_5$ lik 1), som i implementeringen er realisert ved hjelp av 5 akkumulatorer, en adderer og 1 multiplikator.




**Figur 49. Topologi for *hardware*-implementering av DS-modulator.**

## 4.5 PWM

Figur 50 viser prinsippet for hardware-implementeringen av PWM-modulen. Denne består hovedsakelig av et register, en enkel 4-bits teller og en komparator.




**Figur 50. Topologi for *hardware*-implementering av PWM-enhet.**

## 4.6 RAM

Realiseringen av RAM er generert ved hjelp av *Xilinx Core Generator* [36] som ut fra gitte spesifikasjoner kan generere nødvendig nettliste for syntese, samt verilog-kode for simulering.

Størrelsen på de 3 RAM-blokkene i systemet er som følger:

- RAM 1: 24x64 bit
- RAM 2: 24x16 bit
- RAM 3: 24x8 bit

## 4.7 ROM

Filterkoeffisientene for FIR-filtrene i systemet er lagret i ROM. Matlab-modellen for systemet genererer all Verilog-kode med parametere nødvendig for implementering, det vil si i tillegg til koeffisient-verdiene, også informasjon om *fixed point*, bredde på koeffisienter, bredde på adressebuss samt antallet koeffisienter. Dette gjør det enkelt å i ettertid forandre filterspesifikasjonene. Det er imidlertid viktig at antallet koeffisienter for FIR-filtrene er tilpasset størrelsen på RAM benyttet for buffering.

## 4.8 Parametere

Det er mulig å ved hjelp av parametere påvirke implementeringens kompleksitet og ytelse. Det vil her bli gitt en gjennomgang av de viktigste av disse parametrene.

***Databredde***
For modulene før DS-modulatoren er det mulig å spesifisere bredden på inn- og utgangssignalet. Lav databredde vil gi et mindre komplekst design, men introdusere ny kvantiseringsstøy som vil legges til kvantiseringsstøyen som allerede er i inngangssignalet. Bredden på inn- og utgangssignalene til disse modulene bør derfor være noe høyere enn databredden til inngangssignalet for hele systemet. Det bør nevnes at parameteren for bredden til inngangssignalet til FIR-filtrene ikke må overstiger databredden på RAM benyttet til buffering.

***Intern databredde***
For de fleste modulene er det mulig å spesifisere en intern databredde. Dette kan være ønskelig da bredden på mellomregninger teoretisk sett kan overstige bredden på inngangssignalet til modulen.

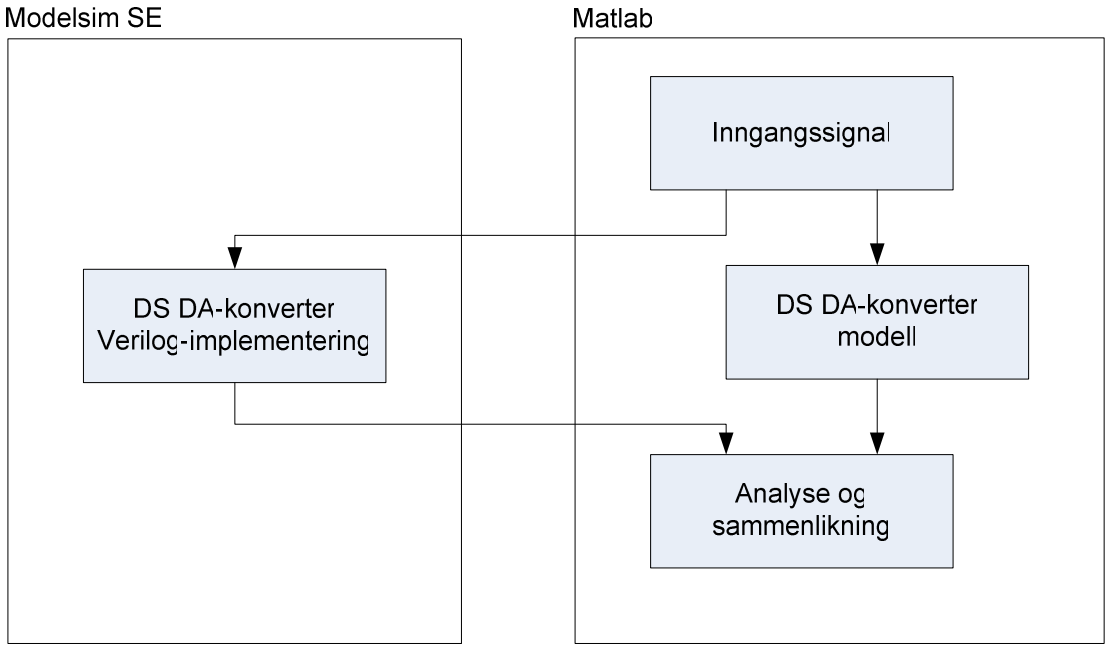
***Intern oppløsning***
For de fleste modulene er det mulig å spesifisere oppløsningen ved interne beregninger. Dette er gunstig da det i noen tilfeller kan være ønskelig at mellomregninger skal ha høyere oppløsning enn utgangssignalet for å unngå et betydelig feil i utgangssignalet som følge av flere avrundinger.

# 5 Simulering

*Det vil i dette kapittelet bli presentert en fullstendig simulering av DA-konverteren. Simuleringer av verilog-kode er utført ved hjelp av Modelsim SE5.7F. Implementeringen i verilog er verifisert med sam-simulering mot Matlab, der simuleringsresultatene fra Modelsim SE5.7F sammenliknes med resultatene fra Matlab-modellen for systemet. Prinsippet for simuleringene er illustrert i Figur 51.*

*Det vil være et avvik mellom verilog-implementeringen og Matlab-modellen da sistnevnte ikke er modellert nøyaktig med tanke på alle begrensningene som introduseres ved hardware-implementering, som eksempelvis nøyaktighet ved bergninger.*

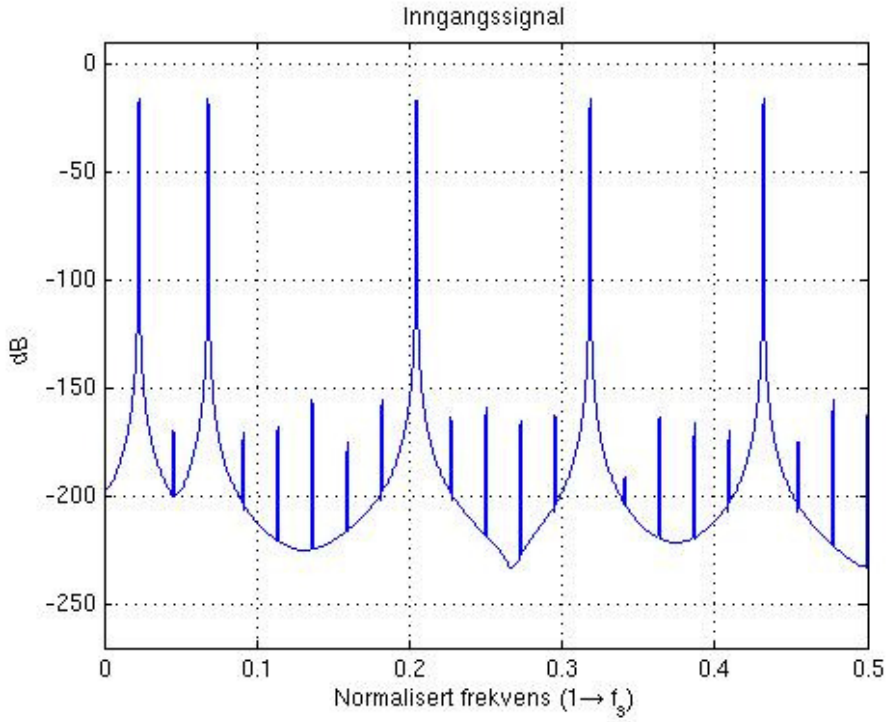
*I 5.1 til 5.5 er det for verilog-implementeringen benyttet 24-bit databredde og 24-bit nøyaktighet ved alle beregninger.*




**Figur 51. Sam-simulering mellom Modelsim SE og Matlab.**

## 5.1 Inngangssignal

Figur 52 viser inngangssignalet for simuleringen. Inngangssignalet består av 5 frekvenser som ved en samplerate på 44.1 kHz vil tilsvare henholdsvis 1, 3, 9, 14, og 19 kHz. Grunnen til at det er valgt et inngangssignal med en sammensetning av frekvenser over hele basisbåndet er at det da er mulig fra figurene å både få et inntrykk av filterkarakteristikker og eventuell støy som måtte introduseres i systemet. Det er benyttet 24-bits kvantisering for inngangssignalet.

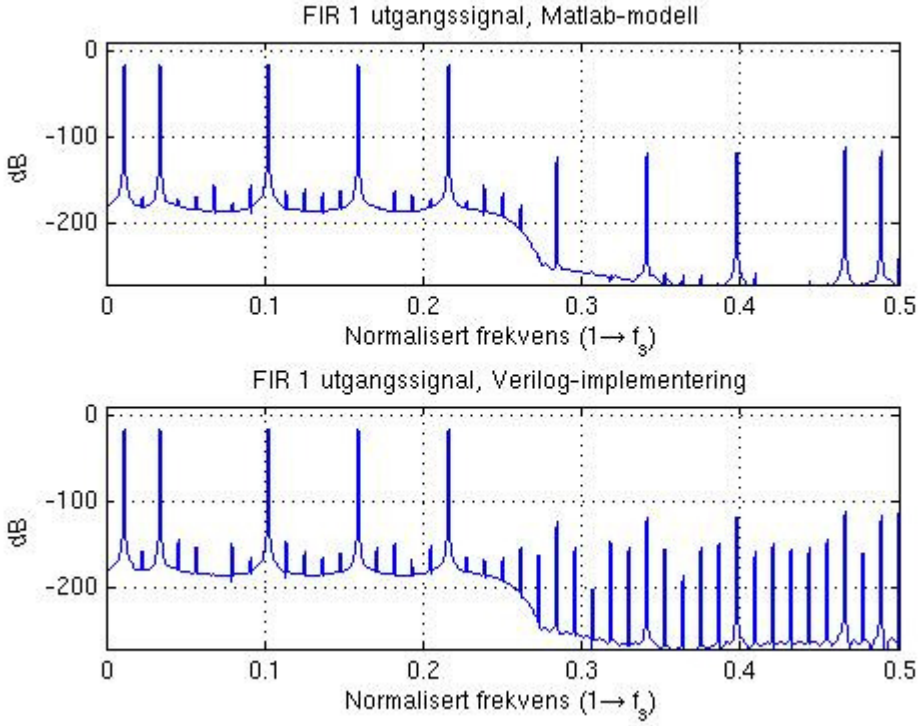



**Figur 52. Inngangssignal for simulering, 0 dBFS.**

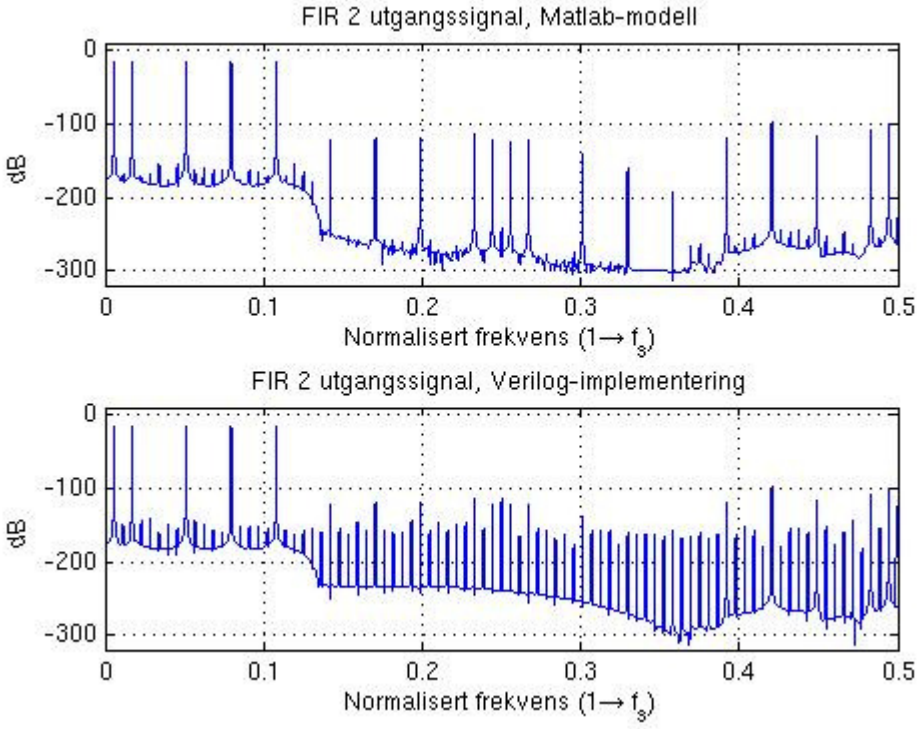
## 5.2 Interpolering og filtrering

Figur 53 til Figur 55 viser utgangssignalet fra de 3 FIR filtrene i systemet, mens Figur 56 viser utgangssignalet fra CIC-filteret. Da utgangssignalet fra de 4 filtrene er oversamplet med en OSR på henholdvis 2, 4, 8 og 32, vil den øvre grensen for basisbåndet tilsvare 0.25, 0.125, 0.0625 og 0.016 normalisert til sampleraten $f_s$.

Fra figurene ser vi tydelig hvordan størrelsen på basisbåndet reduseres i forhold til størrelsen på Nyqust-båndet for hvert interpoleringstrinn. Vi ser også at signalet i Verilog-implementeringen inneholder mer støy enn Matlab-modellen. Dette er ikke uventet da vi vet at både beregningene og signalrepresentasjon i verilog-implementeringen vil ha begrenset nøyaktighet i forhold til Matlab-modellen, noe som vil gi mer kvantiseringsstøy.

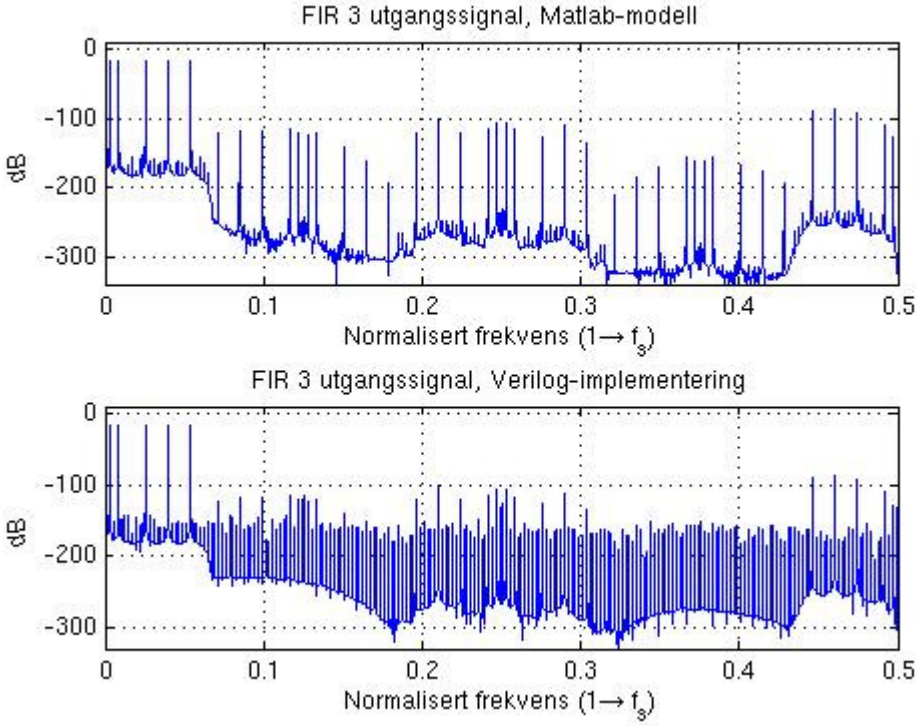



**Figur 53. Utgangssignal FIR-filter 1.**

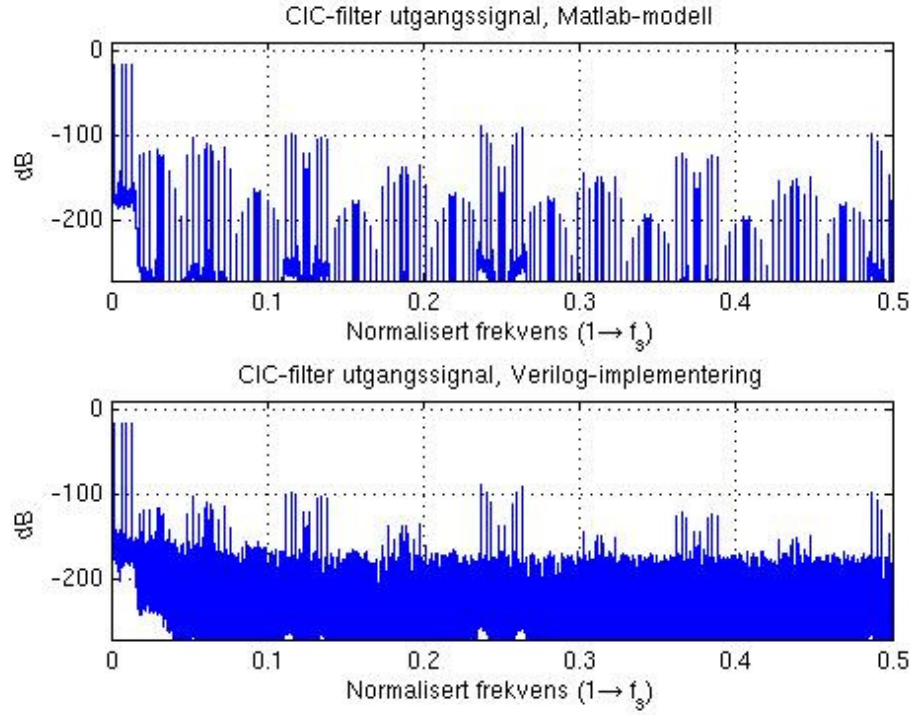



**Figur 54. Utgangssignal FIR-filter 2.**

.



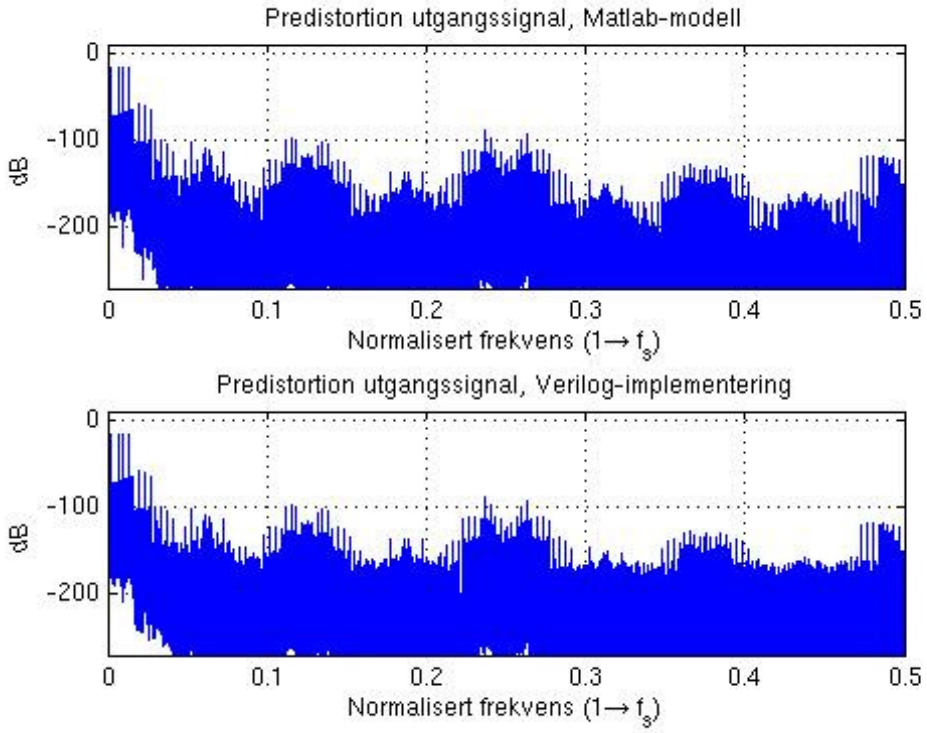

**Figur 55. Utgangssignal FIR-filter 3.**




**Figur 56. Utgangssignal CIC-filter.**

## 5.3 Predistortion

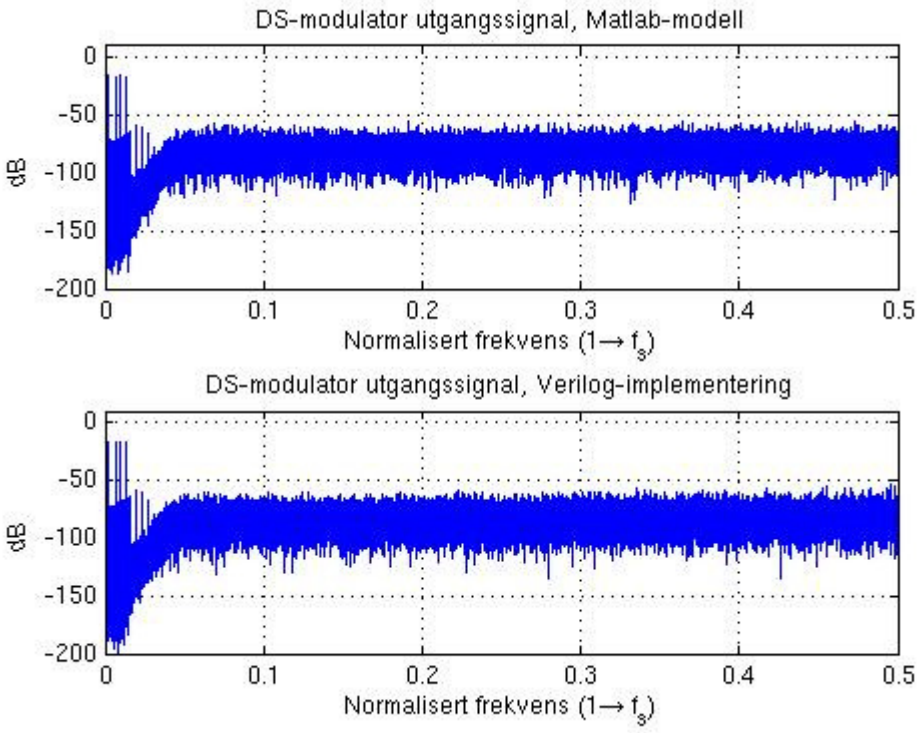
Figur 57 viser utgangssignalet fra predistortion-modulen. Vi ser at dette signalet tilsvarer utgangssignalet fra CIC-filteret, bare med tillegg av betydelig støy. Dette er ikke uventet da vi fra 2.9.3 vet hver enkelt sample blir modifisert i denne, noe som i praksis vil si å tilføre støy.




**Figur 57. Utgangssignal *predistortion*-modul.**

## 5.4 DS-modulator

Figur 58 viser utgangssignalet fra DS-modulatoren. Signalet i dette trinnet er som vi vet kvantisert med 4-bit og støyformet, det vil si at kvantiseringsstøy er flyttet ut av basisbåndet. Dett ser vi tydelig ved at frekvensinnholdet i basisbåndet (fra 0 til 0.016) er tilsvarende som for inngangssignalet til modulen (Figur 52), mens det utenfor basisbåndet er tilført betydelig støy.

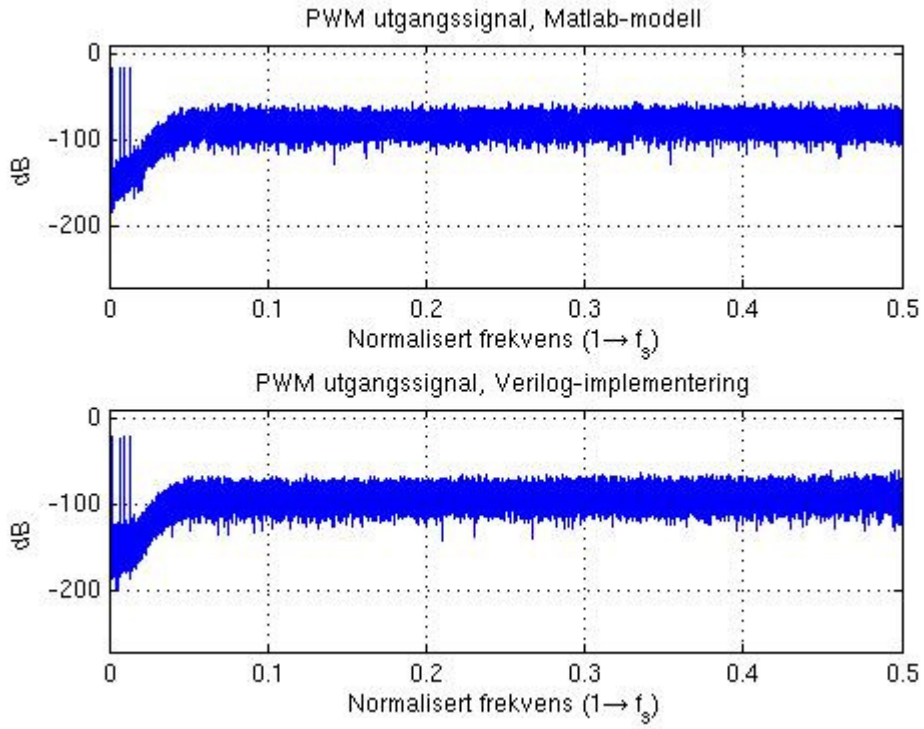



**Figur 58. Utgangssignal DS-modulator.**

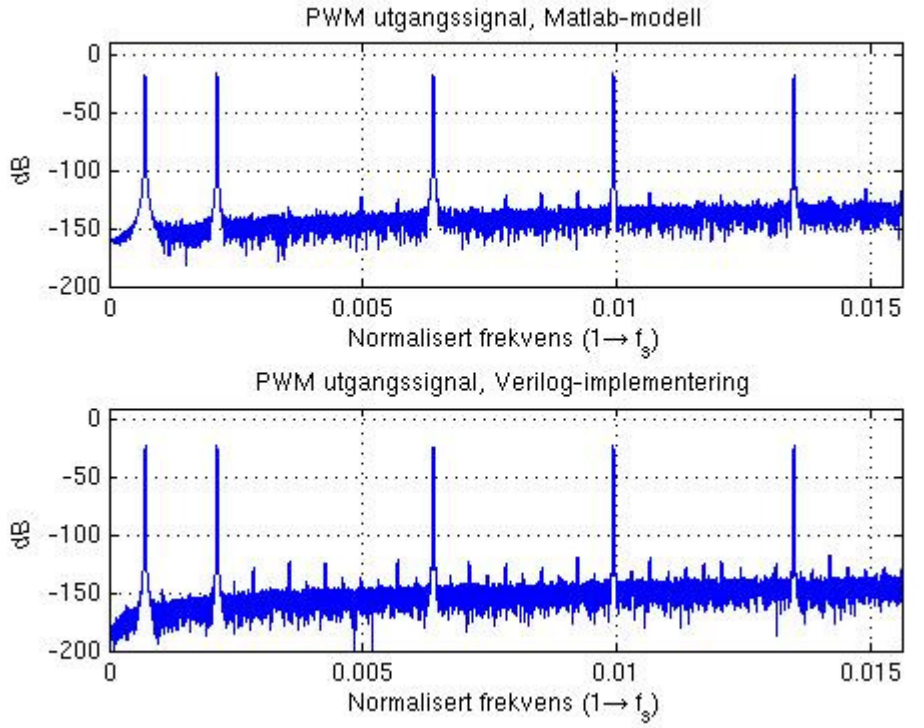
## 5.5 PWM utgangssignal

Figur 59 viser det 1-bis pulsbredde-modulerte utgangssignalet fra systemet. Her ser vi at støykomponentene som ble tilført ved predistortion ”forsvinner” etter pulsbredde-moduleringen.

Figur 60 viser basisbåndet av utgangssignalet i Figur 59, og vi ser at dette inneholder frekvenskomponentene i inngangssignalet.




**Figur 59. PWM utgangssignal, Nyquist-bånd.**




**Figur 60. PWM utgangssignal, basisbånd, 0 dBFS.**

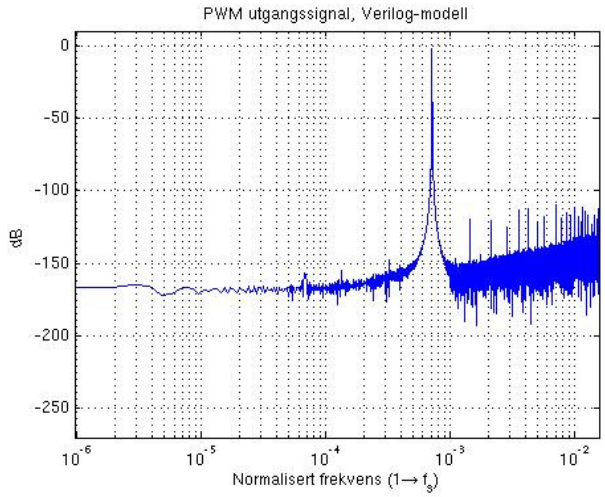
## 5.6 Beregning av THD+N fra Verilog-simulering

Det ble ved dimensjoneringen av systemet beregnet at det med den valgte predistortion-algoritmen skulle være mulig å oppnå et SNR på 121 dB og en THD+N på -101 dB @ $F_s$=44.1 kHz, $F_{sig}$=1 kHz, -1 dBr, 0 dBFS. Denne beregningen tok imidlertid ikke hensyn til eventuell kvantiseringsstøy eller andre avvik som måtte bli introdusert i systemet som følge av begrenset nøyaktighet på interne variabler og beregninger. Eksempelvis bør databredden til signalet mellom de forskjellige modulene være noe høyere enn databredd til inngangssignalet. Vi har sett at hardware-implementeringen tilbyr en rekke parametere for å spesifisere dette ut fra hvilke ressurser som er tilgjengelig eller er ønskelig å bruke.

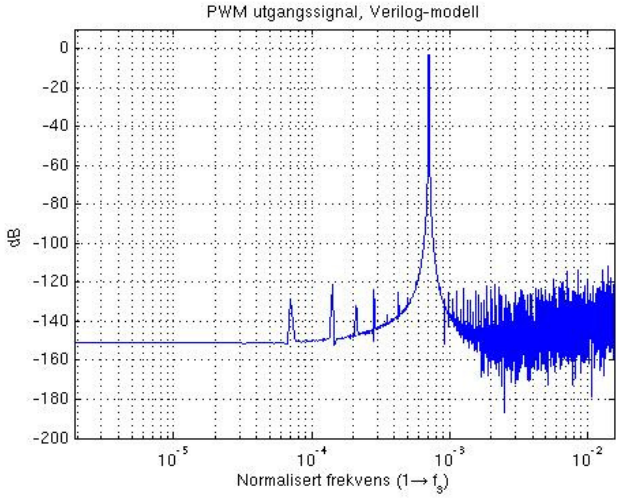
Tabell 4 viser bergnet THD+N med 3 forskjellige oppsett av systemet. Av stabilitetshensyn var det ved disse simuleringene nødvendig å ha høyere nøyaktighet (bit-bredde) internt i DS-modulatoren enn for inngangssignalet til denne. Den interne nøyaktigheten for de resterende modulene tilsvarer nøyaktigheten for inngangssignalene (gitt av kolonne 2 i tabell). Tabellen viser også hvilke krav de forskjellige oppsettene setter til fysiske multiplikatorene i hardware. Vi vil i kapittel 6 se at dette er de mest kritiske ressursbegrensningene i FPGA-kretsen som skal brukes. Figur 61 til Figur 63 viser basisbåndet for utgangssignalet ved de 3 simuleringene. I disse simuleringene ser vi hvordan nivået på støygulvet i utgangssignalet vil variere med bit-bredden i systemet.

| **Oppsett** | **#bit inngangssignal** | **#bit data mellom moduler** | **#bit nøyaktighet internt i DS-modulator** | **Nødvendige multiplikatorer i hardware** | **THD+N @ $F_s$=44.1 kHz, $F_{sig}$=1 kHz, -1.5 dBr, 0 dBFS** |
|---|---|---|---|---|---|
| **1** | 24 | 24 | 24 | 2 stk 24x24<br>1 stk 24x18<br>1 stk 24x15<br>1 stk 24x10 | -98 dB |
| **2** | 16 | 18 | 22 | 1 stk 22x22<br>2 stk 18x18<br>1 stk 18x15<br>1 stk 18x10 | -97.4 dB |
| **3** | 16 | 16 | 22 | 1 stk 22x22<br>1 stk 16x18<br>1 stk 16x16<br>1 stk 16x15<br>1 stk 16x10 | -84.3 dB |

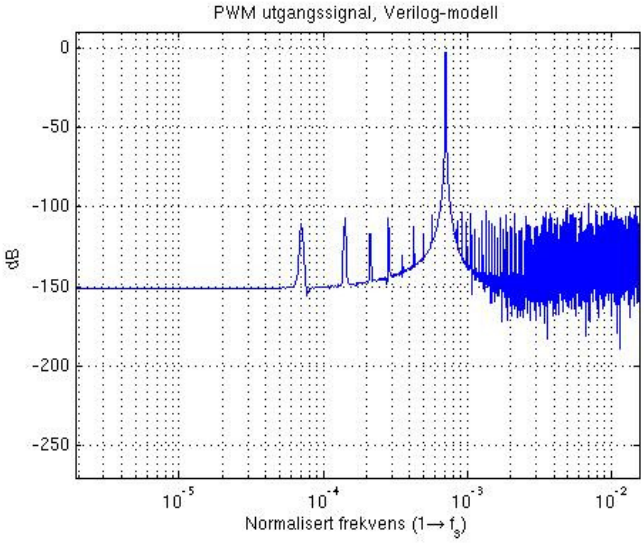
**Tabell 4. Beregnet THD+N fra Verilog-implementering.**




**Figur 61. Verilog-simulering oppsett 1, 24-bit inngangssignal og 24-bit intern databredde.**




**Figur 62. Verilog-simulering oppsett 2, 16-bit inngangssignal og 18-bit intern databredde.**




**Figur 63. Verilog-simulering oppsett 3, 16-bit inngangssignal og 16-bit intern databredde.**

## *5.7 Noen sammenhenger fra simuleringene*

### 5.7.1 Stabilitet og nivå på inngangssignal

Ved simuleringene viser det seg at et fullskala sinusformet inngangssignal vil gi ustabilitet i DS-modulatoren. Dette er grunnen til at det ved alle simuleringer er benyttet et signalnivå 1 til 1.5 dB lavere enn maksimalt signalnivå.

Fra 2.4.5 vet vi at dette nødvendigvis ikke behøver å innebære at det er noe ”galt” med designet av DS-modulatoren, men det er likevel naturlig å undersøke dette litt nærmere.

Vi vet at ustabilitet blant annet kan skyldes en for aggressiv støyforming i DS-modulatoren eller for dårlige interpoleringsfiltre. For å utelukke at dette er tilfelle er det derfor foretatt en simulering med en ”ideell” DS-modulator som oppfyller Lees regel, der det tilføres et perfekt oversamplet signal uten aliasfrekvenser utenfor basisbåndet. Resultatet fra denne simuleringen viser at det også ved det ideelle systemet vil oppstå ustabilitet i DS-modulatoren med et fullskala sinussignal.

Dette kan tyde på at det ikke behøver å være noe i veien med oppsettet for systemet i denne oppgaven. Det kan riktignok tillates en noe høyere signalamplitude for det ideelle systemet (0.5 dB), men dette skyldes nok forskjellen i støy utenfor basisbåndet mellom de to systemene.

Simuleringer med et utvalg av audio-data fra CD viser heller ingen tegn til ustabilitet i DS-modulatoren. Ustabilitet i DS-modulatoren behøver altså ikke å bli et praktisk problem. En løsning på dette kan uansett være å skalere ned inngangssignalet før DS-modulatoren, slik at dette aldri vil overstiger den kritiske verdien for ustabilitet.

### 5.7.2 Stabilitet og intern nøyaktighet i DS-modulator

Fra strukturen for DS-modulatoren i Figur 38 ser vi at det utføres en rekke summasjoner og multiplikasjoner per utgangsverdi. Simuleringer viser at det er nødvendig med en forholdsvis høy intern nøyaktighet i DS-modulatoren for å unngå ustabilitet. Modulatoren synes å være stabil med $\geq 22$ bit intern oppløsning.

# 6 Syntese

*Systemet er syntetisert for en Spartan-3 XC3S200 FPGA ved hjelp av Xilinx ISE 8.1i Project Navigator. Det vil her bli presentert de viktigste synteseresultatene for de 3 oppsettene fra Tabell* 4.

## 6.1 Ressursforbruk

Tabell 5 viser hardware-ressursene de 3 oppsettene vil kreve i FPGA-kretsen. For nærmere forklaring av begrepene i tabellen henvises det til [13].

| Ressurs | #Tilgjengelig i XC3S200 | Oppsett 1 | | Oppsett 2 | | Oppsett 3 | |
|---|---|---|---|---|---|---|---|
| | | #Brukt | % Brukt | #Brukt | % Brukt | #Brukt | % Brukt |
| **Slices** | 1920 | 954 | 49 | 954 | 49 | 933 | 48 |
| **Slice flip flops** | 3840 | 1042 | 27 | 836 | 21 | 807 | 21 |
| **4-input LUTs** | 3840 | 2054 | 53 | 1678 | 43 | 1642 | 42 |
| **Bonded IOBs** | 173 | 28 | 16 | 22 | 12 | 20 | 11 |
| **Block RAMs** | 12 | 3 | 25 | 3 | 25 | 3 | 25 |
| **MULT 18x18** | 12 | 12 | 100 | 8 | 66 | 6 | 50 |
| **GCLKs** | 8 | 5 | 62 | 4 | 50 | 4 | 50 |

**Tabell 5. Ressursbruk i FPGA.**

Fra tabellen ser vi at ingen av de tre oppsettene tilsynelatende overskrider ressursbegrensningene i kretsen.

Det er likevel bare oppsett 1 og 2 som faktisk lar seg syntetisere. Oppsett 3 lar seg ikke syntetisere da det ikke er mulig å bruke alle de 12 multiplikatorene i kombinasjon med 3 RAM-blokker.

Vi ser også at alle oppsettene holder seg godt innenfor resursbegrensningene i kretsen med unntak av multiplikatorantallet, som vil være den begrensende faktoren. Ved oppsett 3 benyttes mindre enn 50% av alle ressurstypene. Det er altså tilsynelatende mulig å implementere 2 instanser av denne kretsen for å få en stereo DA-konverter. Dette ville imidlertid ikke la seg gjøre da man med dette vil få det samme problemet som ved oppsett 1 med tanke på kombinasjon av multiplikatorer og RAM-blokker.

## 6.2 Timinganalyse

Ved syntese blir det beregnet en rekke forskjellige timing-spesifikasjoner for kretsen. Den mest interessante av disse vil i første omgang være kretsens maksimale klokkefrekvens, da denne vil være avgjørende for hvorvidt kretsen kan brukes som planlagt eller ikke. Systemet i

denne oppgaven har 2 klokkeinnganger, det være seg samplefrekvensen $F_s$ og frekvensen $512F_s$. Ved CD-kvalitet lyd vil disse signalene tilsvare henholdsvis 44.1 kHz og 22.6 MHz.

Tabell 6 viser de beregnede maksimale klokkefrekvensene for signalet $512F_s$ ved de 3 oppsettene.

| **Oppsett** | **Maksimal klokkefrekvens for klokkesignal $512F_s$ [MHz]** |
|---|---|
| **1** | 7.2 |
| **2** | 10.2 |
| **3** | 1.64 |

**Tabell 6. Beregnet maksimal klokkefrekvens.**

Fra Tabell 6 vi at ingen av de tre oppsettene vil kunne benyttes med den planlagte sampleraten.

Det er derfor nødvendig å foreta en analyse for finne den kritiske stien i systemet, det vil si den "lengste" logiske veien som er begrensende for den maksimale hastigheten. Synteserapportene viser at denne begrensningen ligger i predistortion-modulen. Fra toplogien for denne i Figur 47 ser vi at den lengste potensielle dataveien er gjennom 1 adderer→1 multiplikator→ 2 multipleksere.

Videre ser vi fra tabellen at klokkefrekvensen noe overraskende er lavest for oppsett 3, som er det minst komplekse av oppsettene. Det er altså tydelig at syntetiseringen ikke er optimal for oppsett 3, og det er derfor heller ikke gitt at dette er tilfell for oppsett 1 og 2. I synteseverktøyet finnes det en rekke muligheter for å påvirke synteseprosessen. Det kan være en mulighet for at ytelsen for systemet kan bedres noe ved å manuelt optimalisere disse syntesepara metrene.

Det vil i kapittel 7 blir diskutert mulige tiltak for å forbedre ytelsen for kretsen.

# 7 Diskusjon og videre arbeid

*Det vil her bli gitt en gjennomgang av sider ved oppgaveløsningen som kunne vært gjort annerledes samt gis konkrete forslag til forbedringer av DA-konverteren.*

## 7.1 Kritisk sti

Fra synteserapportene synes hastigheten for kretsen å være begrenset av for store forsinkelser (for lang kritisk sti) i predistortion-enheten. Topologien i Figur 47 vises at den lengste logiske veien i denne vil inneholde både en adderer og multiplikator i tillegg til flere multipleksere. Dette innebærer at det i løpet av èn klokkeperiode vil måtet utføres både én multiplikasjon og én addisjon i sekvens. Dette er nok ikke en optimal løsning da det burde være mulig å benytte en topologi som utførere disse operasjonene over to klokkesykler.

Denne forbedringen burde også gjøres for DS-modulatoren, som har samme type topologi.

## 7.2 Timing

Fra vedlagt verilog-kode for systemet vil det fremgå at prinsippet for klokking av registre og kombinatorisk logikk er som følger:

- Multipleksere skiftes, og nye beregninger starter på positiv klokkeflanke.
- Skriving av data til registre skjer på negativ klokkeflanke.

Dette er ikke hensiktsmessig da det effektivt vil halvere den mulige klokkefrekvensen. Det burde ikke være en omfattende modifikasjon å forandre implementeringen slik at begge disse operasjonene skjer på samme klokkeflanke.

## 7.3 Forenkling av DS-modulator

Fra topologien for DS-modaulatoren i Figur 38 så vi at denne inneholder 5 akkumulatorer samt én adderer og én multiplikator. En akkumulator består i prinsippet av et register med en tilbakekopling gjennom en adderer. Implementeringen inneholder altså i realiteten 6 adderere, noe som ikke er optimalt med tanke på ressursforbruk.

## 7.4 Registerlevetid

Implementeringen er ikke optimalisert særlig med tanke på registerlevetid og deling av registre. Ved en beregning er det ytterst få variabler som behøver å ”leve” gjennom hele beregningen. Dette vil kunne gi åpning for at flere variabler kan benytte samme fysiske register, bare ved forskjellige tidspunkt. En slik optimalisering vil for det første redusere

antallet registre som benyttes, men også kunne redusere kritisk sti ved at multiplekserene i kretsen blir enklere.

## 7.5 Global deling av hardware-resurser

Implementeringen for DA-konverteren er modulær, hvilket innebærer at hver modul i systemet er implementert som enkeltstående enheter med lokale hardware-ressurser. Dette vil gi en lite effektiv utnyttelse av ressursene da ikke alle modulene bruker "sine" ressursene hele tiden. Det ville være hensiktsmessig å bruke benytte en topologi for systemet som tillater deling av ressurser mellom de forskjellige modulene.

## 7.6 Automatisk generering av kjerner

I dette prosjektet er det meste av verilog-kode bygd opp fra grunnen. Dette er både lærerikt og interessant, men gir nødvendigvis ikke et optimalt resultat da det kreves en del erfaring for å lage gode implementeringer av digitale design i en FPGA-krets. Systemet i denne oppgaven inneholder en del standardenheter, som eksempelvis FIR- og CIC-filter. For implementeringen av disse enhetene kunne det være vel så hensiktsmessig å benytte verktøy for autogenerering av verilog-kode, både med hensyn på kvalitet og arbeidsmengde.

## 7.7 Optimalisering av parametere for verilog-implementering

Fra Tabell 4 så vi at den beste THD+N beregnet fra verilog-simuleringen av systemet er på -98 dB, mens denne fra Matlab-modellen ble beregnet til kunne være -101 dB. Denne forskjellen kan delvis skyldes at inngangssignalet ved Matlab-simuleringen er 0.5 dB høyere enn ved Verilog-simuleringen, men det meste av avviket skyldes nok forskjellen i intern nøyaktighet for de to simuleringene. Det er imidlertid mulig å via parametere spesifisere denne for de forskjellige modulene i verilog-implementeringen. Det ville derfor være interessant å optimalisere disse parametrene ytterligere for å se om en THD+N på -101 dB også er innen rekkevidde for det implementerte systemet, samt finne hvilke hardware-ressurser dette eventuelt ville kreve.

## 7.8 Stabilitet

Det er anbefalt ved design av en delta-sigma-modulator å foreta omfattende simuleringer for å analysere modulatorens stabilitet. I dette prosjektet er det foretatt enkle tester som tyder på at modulatoren er stabil, uten at disse er tilstrekkelig grunnlag for å fastslå dette. Det ville derfor være naturlig å forta ytterlige stabilitetstester.

## 7.9 Modulatorstruktur

Det er i prosjektet lagt ned til dels mye arbeid i studering og simulering av *integral noise shaping* (INS) som presenteres i [7]. Denne modulatorstrukturen skal ettersigende kunne gi en forbedring av THD+N i forhold til den valgte løsningen med CIFB-struktur. Det ville være

interessant å jobbe videre med INS-strukturen i kombinasjon med den mest avanserte av predistortion-algortimene presentert i [10] for å se om DA-konverterens ytelse kunne forbedres ytterligere.

Begge disse teknikkene skal også kunne brukes ved dobbeltsidig PWM, som vil være en ytterligere forbedring av designet.

## 7.10 Passbånd dropp i CIC-filter

Vi har sett at den valgte strukturen for CIC-filteret brukt for interpolering vil ha et passbånd-dropp på 0.1 dB, mens målet for dette ble satt til 0.001 dB. Det bør undersøkes nærmere i hvor stor grad dette vil forringe lydkvaliteten.

Eventuelt kan det forgående FIR-filteret modifiseres til å kompensere for droppet i CIC-filteret.

# 8 Konklusjon

Målet for oppgaven var å implementere en *high-end* delta-sigma DA-konverter i en FPGA.

Ut fra dette ble det i prosjektet utformet konkrete spesifikasjoner for den ferdige DA-konverteren, det være seg en SNR >120 dB og THD+N < -100 dB, samt at DA-konverteren skulle kunne brukes med en samplerate på 44.1 kHz tilsvarende CD-kvalitet.

Ved prosjektslutt foreligger det en implementering som til dels overholder disse spesifikasjonene med tanke på støy (SNR og THD+N), men som ikke overholder spesifikasjonene for samplerate. Det er oppnådd en THD+N på -101 dB for Matlab-modellen av systemet og -98 dB for verilog-implementeringen, mens maksimal samplerate vil være begrenset til rundt 20 kHz. (Fra maksimal frekvens $f_s512$=10.2MHz).

Når det gjelder oppgavens løsning med tanke på oppgavebeskrivelsen, vil nok ikke en DA-konverter med den oppnådde sampleratene kunne klassifiseres som en *high-end* konverter. Når det gjelder THD+N vil det være en diskusjonssak hvorvidt dette kvalifiserer til denne betegnelsen. Vi vet at THD+N ved CD-lyd vil være begrenset til omlag -98 dB, så en forbedring av THD+N ville nok kunne være ønskelig for å unngå at støy som legges til ved DA-omformingen vil være signifikant i forhold til kvantiseringsstøyen i inngangssignalet. *High-end* er uansett et vidt begrep, og en DA-konverter med en THD+N på 98 dB vil nok likevel kunne falle inn under denne betegnelsen.

Selv om resultatet fra oppgaven ikke oppfyller kravet i oppgaveskrivelsen til fulle, føler jeg at prosjektet som helhet har vært meget lærerik og interessant. Ved prosjektstart hadde jeg begrenset erfaring både innen digital signalbehandling, verilog-programmering og bruk av Matlab, som alle vært sentrale momenter ved realiseringen av DA-konverteren. Videre føler jeg at jeg har lært mye om viktigheten av god planlegging og dokumentasjon i et prosjektarbeid som strekker seg over så vidt lang tid.  Dette er noe jeg føler jeg vil dra nytte av ved fremtidige prosjekter.

# 9 Referanser


[1] Richard Schreier, Gabor C. Temes (2005). "*Understanding Delta-sigma Data Converters",* ISBN 0-471-16585-2.

[2] David A. Johns, Ken Martin (1997). "*Analog Integrated Circuit Design*", ISBN0-471-14448-7.

[3] John G. Proakis, Dimitris G. Manolakis. "*Digital Signal Processing"*, ISBN 0-13-394289-9.

[4] Adel S. Sedra, Keneth C. Smith (1998). "*Microelectronic Circuits, fourth edition", ISBN 0-19-511690-9.*

[5] John P. Uyemura. "*Introduction to VLSI Circuits and Systems"*, ISBN 0-471-12704-3.

[6] Matthew P. Donadio (2000). "*CIC Filter Introduction", http://users.snip.net/~donadio/cic.pdf.*

[7] *Pallab* Midya, Matt Miller, Mark Sandler (2000). "*Integral Noise Shaping for Quantization of Pulse Width Modulation"*. Presented at the 109th Convention 2000 September 22-25 Los Angeles, California, USA.

[8] Pallab Midya, Bill Roeckner, Pat Rakers, Poojan Wagh (2000). "*Prediction Correction Algorithm for Natural Pulse Width Modulation",* Presented at the 109th Convention 2000 September 22-25 Los Angeles, California, USA.

[9] Clifford E. Cummings. "*New Verilog Techniques for Creating Parameterized Models"*. http://www.sunburst-design.com/papers/CummingsHDLCON2002_Parameters_rev1_2.pdf

[10] César Pascual, Bill Roeckner (2000). "*Computationally Efficient Conversion from Pulse-Code-Modulation to Naturally-Sampled Pulse-Width-Modulation"*. Presented at the 109th Convention 2000 September 22-25 Los Angeles, California, USA.

[11] César Pascual, Zukui Song, Philip T. Krein, Dilip V. Sarwate, Pallab Midya, William J. Roeckner (2002). "*High-Fidelity PWM Inverter for Audio Amplification Based On Real-Tilme DSP"*. http://www.ifp.uiuc.edu/~sarwate/pubs/Pascual03High.pdf

[12] Asahi Kasei Microsystems (AKM). "*AK4117 Low Power 192 kHz Digital Audio Receiver datasheet"*. http://www.ortodoxism.ro/datasheets2/9/0peyagk0h9la98fjclksxfx5zepy.pdf'

[13] Xilinx. "*Spartan-3E FPGA Family: Complete Data Sheet"*.

*http://direct.xilinx.com/bvdocs/publications/ds312.pdf*

[14] Reto Zimmermann (2006). "*Coding Guidelines for Datapath Synthesis"*. http://www.synopsys.com/products/designware/dwtb/articles/coding_guidelines/coding_guidelines.pdf

[15] Douglas J. Smith (2001). "*HDL Chip Design"*. ISBN 0965193438.

[16] Ivar Løkken (2005). "*High-Resolution Audio DACs Final Report, A Review of the Digital Aidio Conversion Process"*. http://www.iet.ntnu.no/~ivarlo/files/School/PhD/Report_audiodac.pdf

[17] Ivar Løkken (2005). *Interpolation Filters in Delta-sigma DACs.*

[18] Ivar Løkken (2005). "*PCM-PWM analysis brief*". http://www.iet.ntnu.no/~ivarlo/files/School/PhD/03_PCM_to_PWM.pdf

[19] T.S. Doorn, E. van Tuijl, D. Schinkel, A.J. Annema, M. Berkhout, B. Nauta. "*An Audio FIR-DAC in a BCD Process for High Power Class-D Amplifiers"*. http://ieeexplore.ieee.org

[20] Derk Reefman, John van den Homberg, Ed van Tuijl, Corné Bastiaansen, Leon van der Dussen. "*A new Digital-to-Analogue Converter Design Technique for HiFi Applications".* AES Convention Paper 5846, March 2003.

[21] Sanjit A. Mitra, McGraw (2006). "*Digital Signal Processing; A Computer-Based Approach, Third Edition"*. ISBN 0072513780.

[22] Julius O. Smith III. "*Introduction to Digital Filters with Audio Applications"*. http://ccrma.stanford.edu/~jos/filters/

[23] IEEE. *"Std 1364-2001, IEEE standard Verilog Hardware Description Language"*. http://www.ieee.org

[24] Rane*." Pro Audio Reference"*. http://www.rane.com/par-w.html

[25] Nyquist: *"Certain topics in telegraph transmission theory",* Trans. AIEE, vol. 47, pp. 617-644, Apr. 1928.

[26] W. R. Bennett: *"Spectra of Quantized Signals"*, Bell Systems tech journal, vol. 27, pp. 446-472, July 1948.

[27] Risbo, Lars: "*Delta-Sigma Modulators: Stability Analysis and Optimization*", PhD thesis, Technical University of Denmark, June 1994.

[28] Lipshitz, Stanley and Vanderkooy, John: *"Towards a Better Understanding of 1-bit Sigma-Delta Modulators"*, AES Preprint no. 5398.

[29] W. L. Lee, "*A novel higher order interpolative modulator topology for high resolution oversampling A/D converters*" MSc Thesis, Massachusetts Institute of Technology, Cambridge, June 1987.

[30] T. Saramaki, "Design of FIR filters as a tapped cascaded interconnection of identical subfilters," *IEEE Trans. Circuits Syst.*, vol. 34, pp. 1011-1029, Sept. CAS-1987.

[31] Richard Schreier. "*The Delta Sigma Toolbox 7.1*". http://www.mathworks.com/matlabcentral/fileexchange/loadFile.do?objectId=19&objectType=file.

[32] Nielsen, Karsten. "Audio Power Techniques based on efficient Power Conversion". Phd. Thesis, Technical University of Denmark, 1998.

[33] Philps: "Red-Bokk document". Dokument 28/10/04-3122 783 0027 2, 1980.

[34] Rod Elliott. "A Weighting Filter For Audio Measurements". 29. august, 2003. http://sound.westhost.com/project17.htm

[35] "All about DVD", http://www.dvdsoftwareguide.com/all-about-dvd-1-guide.html

[36] Xilinx. "*Core Generator 8.1.03i*". http://www.xilinx.com/ipcenter/coregen/2_1i/docs/ug2.1i.pdf

# 10 Appendiks

## *10.1 Verilog-kode*

### 10.1.1 Toppnivå DA-konverter for Xilinx, med RAM

```
//////////////////////////////////////////////////////////////////////////////////
/////
//
// Project:       Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   Delta-Sigma DA toplevel
//                For XILINX, uses 3 instances of block RAM
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
/////
`timescale 1ns/100ps

module dstoplevel(inputdata, outputdata, sampleclock, sampleclockOSR,
preset);

   `include "coefrom1parameters.v"
   `include "coefrom2parameters.v"
   `include "coefrom3parameters.v"

   // Main parameters
   //parameter DataWidthIn=24;
   parameter DataWidthIn=18;

   input signed [DataWidthIn-1:0] inputdata;
   input sampleclock, sampleclockOSR, preset;
   output outputdata;

   parameter  DataWidthOut=4;
   parameter OSR_clk_period = 1;
   parameter sample_clk_period = 512;

   // Input file parameters
   parameter VectInLength = 1024;

   // Parameters MAC 1
   parameter Mac1InternalFixedPoint=0;
   parameter      Mac1AccWidthExpand=1;

   // Parameters MAC 2
   parameter Mac2InternalFixedPoint=0;
   parameter      Mac2AccWidthExpand=1;

   // Parameters MAC 3
   parameter Mac3InternalFixedPoint=0;
   parameter      Mac3AccWidthExpand=1;
```

```
    // Parameters predistortion module
    parameter PredInternalFixedPoint=0; // InternalFixedPoint
    parameter PredBetaCoef=4; // Corresponds to 1/(2^4)=0.0625

    // Parameters DS module
    //parameter DSInternalWidth=26;
    parameter DSInternalWidth=DataWidthIn+2;
    parameter DSInternalFixedPoint=0;

    // Parameters Ram1
    parameter RAM1_AdrBits=6;
    parameter RAM1_DataWidth=24;

    // Parameters Ram2
    parameter RAM2_AdrBits=4;
    parameter RAM2_DataWidth=24;

    // Parameters Ram3
    parameter RAM3_AdrBits=4;
    parameter RAM3_DataWidth=24;

    wire [8:0] clk_out;

    // General instance wires
    wire signed [DataWidthIn-1:0] macDataIn1;
    wire signed [DataWidthIn-1:0] macDataOut1;
    wire signed [DataWidthIn-1:0] macDataOut2;
    wire signed [DataWidthIn-1:0] macDataOut3;
    wire signed [DataWidthIn-1:0] cicDataOut;
    wire signed [DataWidthIn-1:0] predistortionOut;
    wire [DataWidthOut-1:0] dsDataOut;
    wire pwmDataOut;

    // RAM/ROM wires
    wire [RAM1_AdrBits-1:0] ram1address;
    wire [RAM2_AdrBits-1:0] ram2address;
    wire [RAM3_AdrBits-1:0] ram3address;

    wire [RAM1_DataWidth-1:0] ram1DataIn;
    wire [RAM2_DataWidth-1:0] ram2DataOut;
    wire [RAM3_DataWidth-1:0] ram3DataIn;

    wire [RAM1_DataWidth-1:0] ram1DataOut;
    wire [RAM2_DataWidth-1:0] ram2DataIn;
    wire [RAM3_DataWidth-1:0] ram3DataOut;

    wire [CoefROM1_AdrBits-1:0] CoefROM1_address;
    wire [CoefROM2_AdrBits-1:0] CoefROM2_address;
    wire [CoefROM3_AdrBits-1:0] CoefROM3_address;

    wire [CoefROM1_CoefWidth-1:0] CoefROM1_out;
    wire [CoefROM2_CoefWidth-1:0] CoefROM2_out;
    wire [CoefROM3_CoefWidth-1:0] CoefROM3_out;

    wire ram1clk, ram2clk, ram3clk, ram1En, ram2En, ram3En;

    assign outputdata=pwmDataOut;
    assign macDataIn1=inputdata;

// Module instances
```

```
   coefrom1 rom1 (.CoefROM1_address(CoefROM1_address),
.CoefROM1_out(CoefROM1_out));

   coefrom2 rom2 (.CoefROM2_address(CoefROM2_address),
.CoefROM2_out(CoefROM2_out));

   coefrom3 rom3 (.CoefROM3_address(CoefROM3_address),
.CoefROM3_out(CoefROM3_out));

   ram24x64 ram1 (.addr(ram1address), .clk(ram1clk), .din(ram1DataIn),
.dout(ram1DataOut), .we(ram1En));

   ram24x16 ram2 (.addr(ram2address), .clk(ram2clk), .din(ram2DataIn),
.dout(ram2DataOut), .we(ram2En));

   ram24x8 ram3 (.addr(ram3address), .clk(ram3clk), .din(ram3DataIn),
.dout(ram3DataOut), .we(ram3En));


   clock_module klokke (.preset(preset), .sample_freq(sampleclock),
            .sample_freq512(sampleclockOSR), .fout(clk_out) );

   macmoduleXILINX  #(.DataWidth(DataWidthIn),
                .CoefWidth(CoefROM1_CoefWidth),
                .FmacDivFsample(256),
              .CoefAdrBitlength(CoefROM1_AdrBits),
              .NoCoef(CoefROM1_NoCoef),
              .RamDataWidth(RAM1_DataWidth),
              .RamAdrBitlength(RAM1_AdrBits),
              .InternalFixedPoint(Mac1InternalFixedPoint),
              .AccumulatorWidthExpand(Mac1AccWidthExpand)
              )
            mac1
            (.preset(preset),
             .sample_in_freq(clk_out[0]),
             .ramaddressoutput(ram1address),
                 .ramclkoutput(ram1clk),
                 .ramdatainput(ram1DataOut),
                 .ramdataoutput(ram1DataIn),
                 .ramwrEnoutput(ram1En),
             .mac_freq(clk_out[8]),
             .coefinput(CoefROM1_out),
             .coefaddressoutput(CoefROM1_address),
             .datain(macDataIn1),
             .dataout(macDataOut1));

   macmoduleXILINX  #(.DataWidth(DataWidthIn),
                .CoefWidth(CoefROM2_CoefWidth),
                .FmacDivFsample(128),
            .CoefAdrBitlength(CoefROM2_AdrBits),
            .NoCoef(CoefROM2_NoCoef),
            .RamDataWidth(RAM2_DataWidth),
              .RamAdrBitlength(RAM2_AdrBits),
              .InternalFixedPoint(Mac2InternalFixedPoint),
              .AccumulatorWidthExpand(Mac2AccWidthExpand)
              )
            mac2
            (.preset(preset),
             .sample_in_freq(clk_out[1]),
             .ramaddressoutput(ram2address),
                 .ramclkoutput(ram2clk),
```

```
                .ramdatainput(ram2DataOut),
                .ramdataoutput(ram2DataIn),
                .ramwrEnoutput(ram2En),
             .mac_freq(clk_out[8]),
             .coefinput(CoefROM2_out),
             .coefaddressoutput(CoefROM2_address),
             .datain(macDataOut1),
             .dataout(macDataOut2));

   macmoduleXILINX  #(.DataWidth(DataWidthIn),
                .CoefWidth(CoefROM3_CoefWidth),
                .FmacDivFsample(64),
              .CoefAdrBitlength(CoefROM3_AdrBits),
              .NoCoef(CoefROM3_NoCoef),
              .RamDataWidth(RAM3_DataWidth),
              .RamAdrBitlength(RAM3_AdrBits),
              .InternalFixedPoint(Mac3InternalFixedPoint),
              .AccumulatorWidthExpand(Mac3AccWidthExpand)
              )
            mac3
            (.preset(preset),
             .sample_in_freq(clk_out[2]),
             .ramaddressoutput(ram3address),
                 .ramclkoutput(ram3clk),
                 .ramdatainput(ram3DataOut),
                 .ramdataoutput(ram3DataIn),
                 .ramwrEnoutput(ram3En),
             .mac_freq(clk_out[8]),
             .coefinput(CoefROM3_out),
             .coefaddressoutput(CoefROM3_address),
             .datain(macDataOut2),
             .dataout(macDataOut3));


   cicmodule #(.DataWidth(DataWidthIn)) cic (.inputclk(clk_out[3]),
.internalclk(clk_out[8]),
               .outputclk(clk_out[5]), .preset(preset),
.datainput(macDataOut3), .dataoutput(cicDataOut) );


   predistortionmoduleA #(.DataWidth(DataWidthIn),
.InternalFixedPoint(PredInternalFixedPoint), .BetaCoef(PredBetaCoef) )
predistortion
                         (.inputdata(cicDataOut),
.outputdata(predistortionOut),
                          .inputclk(clk_out[5]),
.inputclkx16(sampleclockOSR),
                          .preset(preset));

   dsmoduleCIFB #(.DataWidthIn(DataWidthIn),
.DataWidthOut(DataWidthOut),.InternalWidth(DSInternalWidth),
.InternalFixedPoint(DSInternalFixedPoint)) dsCIFB
            (.inputdata(predistortionOut), .outputdata(dsDataOut),
.inputclk(clk_out[5]),
             .inputclkx16(sampleclockOSR), .preset(preset));

   pwmmodule #(.InputWidth(DataWidthOut)) pwm (.inputdata(dsDataOut),
.outputdata(pwmDataOut),
                     .inputclk(clk_out[5]), .outputclk(sampleclockOSR),
.preset(preset))
endmodule
```

### 10.1.2 FIR MAC interpolator for RAM (Xilinx)

```
//////////////////////////////////////////////////////////////////////////////////
//
//
// Project:       Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   MAC implementation, upsample x2 + halfband FIR
//                Made for XILINX. Interfaces to RAM in different module
//                for buffering.
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module macmoduleXILINX(preset, sample_in_freq, ramaddressoutput,
ramclkoutput, ramdatainput, ramdataoutput, ramwrEnoutput, coefinput,
coefaddressoutput, mac_freq, datain, dataout);
   // Most parameters will in practice be passed from higher level module
   parameter DataWidth=24;
   parameter RamDataWidth=24;
   parameter CoefWidth=21;
   parameter CoefAdrBitlength=5;
   parameter NoCoef=32;
   parameter FmacDivFsample=256;

   parameter RamAdrBitlength=CoefAdrBitlength+1;

   // Requirements: mac_freq > sample_infreq*4*NoCoef+1
   // Note: Coef fixed point = CoefWidth-1
   // Accumulator width = DataWidth + AccumulatorWidthExpand +
InternalFixedPoint

   parameter InternalFixedPoint=8;
   parameter AccumulatorWidthExpand=4;

   input preset, sample_in_freq, mac_freq;
   output [RamAdrBitlength-1:0] ramaddressoutput;
   output ramclkoutput, ramwrEnoutput;
   input signed [RamDataWidth-1:0] ramdatainput;
   output signed [RamDataWidth-1:0] ramdataoutput;

   wire signed [DataWidth-1:0] internalramdatainput;

   input signed [CoefWidth-1:0] coefinput;
   input signed [DataWidth-1:0] datain;

   output [CoefAdrBitlength-1:0] coefaddressoutput;
   output signed [DataWidth-1:0] dataout;

   reg ramwrEn, accEn;

   reg [CoefAdrBitlength:0] ramzpointer; // Ramzpointer must roll round
   reg [CoefAdrBitlength:0] ramindex;

   wire signed [DataWidth+InternalFixedPoint-1:0]addIn;
```

```
   reg signed [DataWidth+InternalFixedPoint+AccumulatorWidthExpand-1:0]
accReg;

   wire signed [DataWidth+CoefWidth-1:0] multOut;
   reg signed [DataWidth-1:0] macOut;

   reg [CoefAdrBitlength-1:0]coefaddress;
   reg [CoefAdrBitlength+4:0] statecount;

   assign ramdataoutput=datain;
   assign internalramdatainput=ramdatainput;
   assign multOut=internalramdatainput*coefinput;

   assign addIn=multOut>>>(CoefWidth-InternalFixedPoint-1);

   assign ramwrEnoutput=ramwrEn;
   assign
ramaddressoutput=(ramzpointer+ramindex)&{(CoefAdrBitlength+1){1'b1}};
   assign ramclkoutput=mac_freq;
   assign dataout=macOut;
   assign coefaddressoutput=coefaddress;

   always @ (posedge preset or posedge mac_freq)
      begin
         if(preset==1)
            ramzpointer=0;
         else
            if(statecount == (FmacDivFsample-2))
               ramzpointer=ramzpointer-1;
         end

   //Accumulate
   always @ (posedge preset or posedge mac_freq or posedge sample_in_freq)
      begin
         if(preset==1) begin
            accReg=0;
          statecount=0;

       end
         else if (sample_in_freq==1) begin
            accReg=0;
            statecount=0;
         end
       else begin
          if(accEn==1)
             accReg=accReg+addIn;
       statecount=statecount+1;
         end
      end

   // Combinatorial logic
   always @ (statecount or accReg or coefinput or internalramdatainput)
      begin
            if(statecount < NoCoef) begin

                coefaddress=statecount;
                ramindex=statecount;
                ramwrEn=0;
                accEn=1;
              end
              else
```

```
                if(statecount < (NoCoef*2)) begin
                   coefaddress=(NoCoef*2-1)-statecount;
                   ramindex=statecount;
                   ramwrEn=0;
                   accEn=1;
                  end
                  else if(statecount < (FmacDivFsample-1)) begin   // If
last-1
                   coefaddress=(NoCoef*2-1)-statecount; // Don't care
                   ramindex=NoCoef-2;
                   ramwrEn=0;                          // Don't care
                   accEn=0;
                  end
                  else begin                           // last step
                   coefaddress=(NoCoef*2-1)-statecount; // Don't care
                   ramindex=NoCoef*2-1;
                   ramwrEn=1;                           // Don't care
                   accEn=0;
                  end

           if(statecount < (FmacDivFsample/2))
             macOut=accReg>>>InternalFixedPoint;
          else
             macOut=internalramdatainput;
     end
endmodule
```

### 10.1.3 ROM for instans 1 av FIR MAC

```
module coefrom1(CoefROM1_address, CoefROM1_out);

   input [4:0] CoefROM1_address;
   output signed [17:0] CoefROM1_out;

   reg signed [17:0] CoefROM1_outreg;
   assign CoefROM1_out=CoefROM1_outreg;


   always @ (CoefROM1_address)
   begin
      case(CoefROM1_address)
         5'd0: CoefROM1_outreg=-18'sd3;
         5'd1: CoefROM1_outreg=18'sd5;
         5'd2: CoefROM1_outreg=-18'sd10;
         5'd3: CoefROM1_outreg=18'sd16;
         5'd4: CoefROM1_outreg=-18'sd26;
         5'd5: CoefROM1_outreg=18'sd39;
         5'd6: CoefROM1_outreg=-18'sd58;
         5'd7: CoefROM1_outreg=18'sd83;
         5'd8: CoefROM1_outreg=-18'sd115;
         5'd9: CoefROM1_outreg=18'sd157;
         5'd10: CoefROM1_outreg=-18'sd211;
         5'd11: CoefROM1_outreg=18'sd278;
         5'd12: CoefROM1_outreg=-18'sd361;
         5'd13: CoefROM1_outreg=18'sd462;
         5'd14: CoefROM1_outreg=-18'sd585;
         5'd15: CoefROM1_outreg=18'sd734;
         5'd16: CoefROM1_outreg=-18'sd912;
         5'd17: CoefROM1_outreg=18'sd1124;
         5'd18: CoefROM1_outreg=-18'sd1376;
         5'd19: CoefROM1_outreg=18'sd1675;
         5'd20: CoefROM1_outreg=-18'sd2031;
         5'd21: CoefROM1_outreg=18'sd2456;
         5'd22: CoefROM1_outreg=-18'sd2967;
         5'd23: CoefROM1_outreg=18'sd3591;
         5'd24: CoefROM1_outreg=-18'sd4365;
         5'd25: CoefROM1_outreg=18'sd5353;
         5'd26: CoefROM1_outreg=-18'sd6663;
         5'd27: CoefROM1_outreg=18'sd8502;
         5'd28: CoefROM1_outreg=-18'sd11313;
         5'd29: CoefROM1_outreg=18'sd16250;
         5'd30: CoefROM1_outreg=-18'sd27549;
         5'd31: CoefROM1_outreg=18'sd83354;
      endcase
   end
endmodule
```

### 10.1.4 ROM for instans 2 av FIR MAC

```
module coefrom2(CoefROM2_address, CoefROM2_out);

   input [2:0] CoefROM2_address;
   output signed [14:0] CoefROM2_out;

   reg signed [14:0] CoefROM2_outreg;
   assign CoefROM2_out=CoefROM2_outreg;


   always @ (CoefROM2_address)
   begin
      case(CoefROM2_address)
         3'd0: CoefROM2_outreg=-15'sd1;
         3'd1: CoefROM2_outreg=15'sd7;
         3'd2: CoefROM2_outreg=-15'sd40;
         3'd3: CoefROM2_outreg=15'sd151;
         3'd4: CoefROM2_outreg=-15'sd450;
         3'd5: CoefROM2_outreg=15'sd1147;
         3'd6: CoefROM2_outreg=-15'sd2811;
         3'd7: CoefROM2_outreg=15'sd10188;
      endcase
   end
endmodule
```

### 10.1.5 ROM for instans 3 av FIR MAC

```
module coefrom3(CoefROM3_address, CoefROM3_out);

   input [1:0] CoefROM3_address;
   output signed [9:0] CoefROM3_out;

   reg signed [9:0] CoefROM3_outreg;
   assign CoefROM3_out=CoefROM3_outreg;


   always @ (CoefROM3_address)
   begin
      case(CoefROM3_address)
         2'd0: CoefROM3_outreg=-10'sd2;
         2'd1: CoefROM3_outreg=10'sd14;
         2'd2: CoefROM3_outreg=-10'sd63;
         2'd3: CoefROM3_outreg=10'sd307;
      endcase
   end
endmodule
```

### 10.1.6 Parameter for ROM 1

```
parameter CoefROM1_CoefWidth=18;
parameter CoefROM1_NoCoef=32;
parameter CoefROM1_AdrBits=5;
```

### 10.1.7 Parametere for ROM 2

```
parameter CoefROM2_CoefWidth=15;
parameter CoefROM2_NoCoef=8;
parameter CoefROM2_AdrBits=3;
```

### 10.1.8 Parametere for ROM 3

```
parameter CoefROM1_CoefWidth=18;
parameter CoefROM1_NoCoef=32;
parameter CoefROM1_AdrBits=5;
```

### 10.1.9 CIC-filter

```
//////////////////////////////////////////////////////////////////////////////
//
//
// Project:             Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   CIC interpolator implementation, 3 stages
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////

// internalclk >= 8*outputclk

module cicmodule
               (
                inputclk,
                internalclk,
                outputclk,
                preset,
                datainput,
                dataoutput
                );
  parameter DataWidth=18;

  parameter OutputDiv=4; // OutputDiv=(order-
1)*ln(outputclk/inputclk)/ln(2)

  input   inputclk;
  input   internalclk;
  input   outputclk;
  input   preset;
  input   signed [(DataWidth-1):0] datainput;
  output  signed [(DataWidth-1):0] dataoutput;

  // Registers
  reg inputclkperiod;
  reg [2:0] statecount;
  reg  signed [(DataWidth-1):0] inputreg;

  //   -- Stage 1 (comb) signals, gain=2 -> + 1 bit
  reg  signed [DataWidth:0] int0;

  //   -- Stage 2 (comb) signals, gain=2 -> + 1 bit
  reg  signed [DataWidth+1:0] int1;

  //   -- Stage 3 (comb) signals, gain=2 -> + 1 bit
  reg  signed [DataWidth+2:0] int2;
  reg  signed [DataWidth+2:0] tempreg0;
  reg  signed [DataWidth+2:0] tempreg1;


  //   -- Stage 4 (integrator) signals, gain=2 -> + 1 bit
  reg  signed [DataWidth+4:0] int3;

  //   -- Stage 5 (integrator) signals, gain=0.5 -> -1 bit
```

```
reg  signed [DataWidth+5:0] int4;

//   -- Stage 6 (integrator) signals, gain=4 -> +2 bit
reg  signed [DataWidth+7:0] int5;


reg  signed [DataWidth+7:0] addInputA;
reg  signed [DataWidth+7:0] addInputB;
reg  signed [DataWidth+7:0] addInputBsig;
reg addsubSelect;
wire signed [DataWidth+7:0] addOutput;

// Add/sub
always@(addInputB or addsubSelect)
    begin
       if(addsubSelect)
          addInputBsig=-addInputB;
       else
          addInputBsig=addInputB;
    end

// Add
assign addOutput=addInputA+addInputBsig;

// Output assign
assign dataoutput=int5>>>OutputDiv;

// Clk operations
always @ (posedge inputclk or posedge outputclk or posedge preset)
   begin
      if(preset==1)
         inputclkperiod=1;
          else if(inputclk==1)
         inputclkperiod=1;
            else
         inputclkperiod=0;
   end


always @ (posedge inputclk or posedge preset)
   begin
      if (preset == 1'b1)
         inputreg = 0;
      else
         inputreg = datainput;
   end

always @ (posedge internalclk or posedge inputclk or posedge preset)
   begin
      if(preset==1)
         statecount=0;
           else if(inputclk==1)
               statecount=0;
      else
            statecount=statecount+1;
   end

always @ (negedge internalclk or posedge preset)
    begin
       if(preset==1 ) begin
          tempreg0=0;
```

```
            tempreg1=0;
            int0=0;
            int1=0;
            int2=0;
            int3=0;
            int4=0;
            int5=0;
         end
            else begin
               case(statecount)
               3'd0: begin
                     if(inputclkperiod==1) begin
                        tempreg0=addOutput;
                        int0=inputreg;
                     end
                  end
               3'd1: begin
                     if(inputclkperiod==1) begin
                        tempreg1=addOutput;
                        int1=tempreg0;
                     end
                  end
               3'd2: begin
                     if(inputclkperiod==1) begin
                        int2=tempreg1;
                        tempreg0=addOutput;
                     end
                  end
               3'd3: begin
                     int3=addOutput;
                  end
               3'd4: begin
                     int4=addOutput;
                  end
               3'd5: begin
                     int5=addOutput;
                  end
              endcase
          end
       end

   // Combinatorial logic
   always @ (statecount,inputreg, int0, int1, int2, int3, int4, int5,
tempreg0, tempreg1, inputclkperiod)
      begin
         case(statecount)
            3'd0: begin
                     addInputA=inputreg;
                     addInputB=int0;
                     addsubSelect=1;
                   end
            3'd1: begin
                     addInputA=tempreg0;
                     addInputB=int1;
                     addsubSelect=1;
                  end
            3'd2: begin
                     addInputA=tempreg1;
                     addInputB=int2;
                     addsubSelect=1;
                  end
```

```
            3'd3: begin
                     addInputA=int3;
                     addInputB = (inputclkperiod == 1'b1) ? tempreg0 : 0;
// Upsampling
                       addsubSelect=0;
                    end
            3'd4: begin
                     addInputA=int4;
                     addInputB=int3;
                     addsubSelect=0;
                   end
            3'd5: begin
                     addInputA=int5;
                     addInputB=int4;
                     addsubSelect=0;
                  end
        endcase
       end

endmodule
```

### 10.1.10 Predistortion

```
//////////////////////////////////////////////////////////////////////////////
/////
//
// Project:             Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   Transforms PCM input data to natural PWM duty ratios
(signed output)
//
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
/////


module predistortionmoduleA(inputdata, outputdata, inputclk, inputclkx16,
preset);
   // All parameteres will in practice be passed from a higher level module
   parameter DataWidth=24;
   parameter InternalFixedPoint=4;
   parameter BetaCoef=4; // Corresponds to 1/(2^4)=0.0625
   parameter InternalWidth=DataWidth+InternalFixedPoint+3;
   // Note: Multipliers will be sized InternalWidth*InternalWidth

   parameter signed [InternalWidth-1:0] ZeroConst= 0;

   input   signed [DataWidth-1:0] inputdata;
   output  signed [DataWidth-1:0] outputdata;
   input   inputclk;
   input   inputclkx16;
   input   preset;

   parameter UnitConst=1<<<(DataWidth+InternalFixedPoint-1);

   reg signed [InternalWidth-1:0] addInputA;
   reg signed [InternalWidth-1:0] addInputB;
   reg signed [InternalWidth-1:0] addInputBsig;
   reg addsubSelect;
   wire signed [InternalWidth-1:0] addOutput;

   reg signed [InternalWidth-1:0] multInputA;
   reg signed [InternalWidth-1:0] multInputB;
   wire signed [(InternalWidth*2)-1:0] multOutput;
   wire signed [(InternalWidth-1)-1:0] multOutputFixed;

   reg signed [InternalWidth-1:0] slope;
   reg signed [InternalWidth-1:0] slopePow;
   reg signed [InternalWidth-1:0] denum;
   reg signed [InternalWidth-1:0] tempsum;
   reg signed [InternalWidth-1:0] parab;
   reg signed [InternalWidth-1:0] tx;

   reg signed [InternalWidth-1:0] inputreg0;
   reg signed [InternalWidth-1:0] inputreg1;
   reg signed [InternalWidth-1:0] inputreg2;
   reg signed [InternalWidth-1:0] inputreg3;
```

```
reg signed [DataWidth-1:0] outputreg;

reg [3:0] statecount;

// Add/sub select
always@(addInputB or addsubSelect)
   begin
      if(addsubSelect)
         addInputBsig=-addInputB;
      else
         addInputBsig=addInputB;
   end

// Add/mult
assign multOutput=multInputA*multInputB;
assign multOutputFixed=multOutput>>>(DataWidth+InternalFixedPoint-1);

assign addOutput=addInputA+addInputBsig;
assign outputdata=outputreg;

always @ (posedge inputclkx16 or posedge inputclk or posedge preset)
   begin
      if(preset==1)
         statecount=ZeroConst;
      else if(inputclk==1)
         statecount=ZeroConst;
      else
         statecount=statecount+1;
   end

// Register transfers on negedge
always @ (negedge inputclkx16 or posedge preset)
   begin
      if(preset==1 ) begin
         slope=ZeroConst;
         slopePow=ZeroConst;
         denum=ZeroConst;
         tempsum=ZeroConst;
         parab=ZeroConst;
         tx=ZeroConst;
         outputreg=0;
      end
         else begin
            case(statecount)
            4'd0: begin
                  slope=addOutput>>>1;
               end
            4'd1: begin
                  denum=addOutput;
                  slopePow=multOutputFixed;
               end
            4'd2: begin
                  denum=addOutput;
                  slopePow=multOutputFixed;
               end
            4'd3: begin
                  denum=addOutput;
                  slopePow=multOutputFixed;
               end
            4'd4: begin
```

```
                    denum=addOutput;
                    slopePow=multOutputFixed;
                end
            4'd5: begin
                    denum=addOutput;
                    slopePow=multOutputFixed;
                end
            4'd6: begin
                    denum=addOutput;
                    slopePow=multOutputFixed;
                end
            4'd7: begin
                    denum=addOutput;
                    slopePow=multOutputFixed;
                end
            4'd8: begin
                    denum=addOutput;
                    slopePow=multOutputFixed;
                end
            4'd9: begin
                    denum=addOutput;
                end
            4'd10: begin
                    tempsum=addOutput;
                    tx=multOutputFixed;
                end
            4'd11: begin
                    parab=addOutput;
                end
            4'd12: begin
                    tempsum=addOutput;
                end
            4'd13: begin
                   tempsum=multOutputFixed;
                end
            4'd14: begin
                    outputreg=addOutput>>>InternalFixedPoint;
                end
         endcase
      end
   end

   // Combinatorial logic
   always @ (statecount, inputreg0, inputreg1, inputreg2, inputreg3, tx,
slope, slopePow, denum, addOutput,
             multOutputFixed, tempsum, parab)
      begin
         case(statecount)
            4'd0: begin
                    addInputA=inputreg2;
                    addInputB=inputreg1;
                    addsubSelect=1;
                    multInputA=0;
                    multInputB=0;
                  end
            4'd1: begin
                    addInputA=UnitConst;
                    addInputB=slope;
                    addsubSelect=0;
                    multInputA=slope;
                    multInputB=slope;
```

```
      end
4'd2: begin
         addInputA=denum;
         addInputB=slopePow;
         addsubSelect=0;
         multInputA=slopePow;
         multInputB=slope;
      end
4'd3: begin
         addInputA=denum;
         addInputB=slopePow;
         addsubSelect=0;
         multInputA=slopePow;
         multInputB=slope;
      end
4'd4: begin
         addInputA=denum;
         addInputB=slopePow;
         addsubSelect=0;
         multInputA=slopePow;
         multInputB=slope;
      end
4'd5: begin
         addInputA=denum;
         addInputB=slopePow;
         addsubSelect=0;
         multInputA=slopePow;
         multInputB=slope;
      end
4'd6: begin
         addInputA=denum;
         addInputB=slopePow;
         addsubSelect=0;
         multInputA=slopePow;
         multInputB=slope;
     end
4'd7: begin
         addInputA=denum;
         addInputB=slopePow;
         addsubSelect=0;
         multInputA=slopePow;
         multInputB=slope;
      end
4'd8: begin
         addInputA=denum;
         addInputB=slopePow;
         addsubSelect=0;
         multInputA=slopePow;
         multInputB=slope;
      end
4'd9: begin
         addInputA=denum;
         addInputB=slopePow;
         addsubSelect=0;
         multInputA=slopePow;
         multInputB=slope;
      end
4'd10: begin
         addInputA=inputreg1;
         addInputB=inputreg2;
         addsubSelect=0;
```

```
                    multInputA=denum;
                    multInputB=addOutput>>>1;
                end
            4'd11: begin
                    addInputA=UnitConst;
                    addInputB=multOutputFixed;
                    addsubSelect=1;
                    multInputA=tx;
                    multInputB=tx;
                end
            4'd12: begin
                    addInputA=tempsum;
                    addInputB=inputreg0;
                    addsubSelect=1;
                    multInputA=0;
                    multInputB=0;
                end
            4'd13: begin
                    addInputA=tempsum;
                    addInputB=inputreg3;
                    addsubSelect=1;
                    multInputA=parab;
                    multInputB=addOutput>>>BetaCoef;
                end
            4'd14: begin
                    addInputA=tempsum;
                    addInputB=tx;
                    addsubSelect=0;
                    multInputA=0;
                    multInputB=0;
                end
              default: begin
                    addInputA=0;
                    addInputB=0;
                    addsubSelect=0;
                    multInputA=0;
                    multInputB=0;
                end
        endcase
   end

  // Read inputdata and shift inputbuffer
  always @ (posedge inputclk or posedge preset)
      begin
         if(preset==1)
         begin
            inputreg0= 0;
            inputreg1= 0;
            inputreg2= 0;
            inputreg3= 0;
         end
         else
         begin
            inputreg0=inputreg1;
            inputreg1=inputreg2;
            inputreg2=inputreg3;
            inputreg3=inputdata<<<InternalFixedPoint;
         end
      end
endmodule
```

### 10.1.11 DS-modulator CIFB

```
//////////////////////////////////////////////////////////////////////////////
/////
//
// Project:             Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   5. order DS modulator using CIFB structure
//
// additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
/////



module dsmoduleCIFB (inputdata, outputdata, inputclk, inputclkx16, preset);
   parameter DataWidthIn=16;
   parameter DataWidthOut=4;
   parameter InternalWidth=32;  // 32 bit resolution
   parameter InternalFixedPoint=4; // InternalFixedPoint

   // InternalWidth must be > max(Coeflength,DataWidthIn,
(DataWidthIn+InternalFixedPoint))

  parameter CoefFixedPoint = 17;
  parameter Coeflength=CoefFixedPoint+1;

  // Accurate Coef values 32 bit fixedpoint
  parameter signed [Coeflength-1:0] A0 = -'sd173;
  parameter signed [Coeflength-1:0] A1 = -'sd2212;
  parameter signed [Coeflength-1:0] A2 = -'sd14700;
  parameter signed [Coeflength-1:0] A3 = -'sd54540;
  parameter signed [Coeflength-1:0] A4 = -'sd123237;

  parameter signed [Coeflength-1:0] B0 = 'sd173;
  parameter signed [Coeflength-1:0] B1 = 'sd2212;
  parameter signed [Coeflength-1:0] B2 = 'sd14700;
  parameter signed [Coeflength-1:0] B3 = 'sd54540;
  parameter signed [Coeflength-1:0] B4 = 'sd123237;

   parameter signed [Coeflength-1:0] G0 = -'sd366;
   parameter signed [Coeflength-1:0] G1 = -'sd1035;

   input   signed [DataWidthIn-1:0] inputdata;
   output  [DataWidthOut-1:0] outputdata;
   input   inputclk;
   input   inputclkx16;
   input   preset;

   reg  signed [InternalWidth-1:0] acc0;
   reg  signed [InternalWidth-1:0] acc1;
   reg  signed [InternalWidth-1:0] acc2;
   reg  signed [InternalWidth-1:0] acc3;
   reg  signed [InternalWidth-1:0] acc4;

   //wire  signed [InternalWidth-1:0] quantout;
```

```
   reg  signed [InternalWidth-1:0] quantout;
   reg signed [InternalWidth-1:0] inputreg;
   reg signed [InternalWidth-1:0] outputreg;

   reg signed [InternalWidth-1:0] accInput0;
   reg signed [InternalWidth-1:0] accInput1;
   reg signed [InternalWidth-1:0] accInput2;
   reg signed [InternalWidth-1:0] accInput3;
   reg signed [InternalWidth-1:0] accInput4;

   reg signed [Coeflength-1:0] multInputA;
   reg signed [InternalWidth-1:0] multInputB;
   wire signed [(Coeflength+InternalWidth)-1:0] multOutFixed;
   wire signed [InternalWidth-1:0] multOut;

   reg signed [InternalWidth-1:0] addInputA;
   reg signed [InternalWidth-1:0] addInputB;
   wire signed [InternalWidth-1:0] addOut;

   parameter maxvalue = 2**(DataWidthIn+InternalFixedPoint-1)-1;
   parameter minvalue = -2**(DataWidthIn+InternalFixedPoint-1);


   reg [3:0] statecount;

      always @ (outputreg)
      begin

   //Output quantize

         if(outputreg > maxvalue) begin
            quantout=maxvalue;
         end
         else
            if(outputreg < minvalue) begin
               quantout=minvalue;
         end
         else begin
            quantout=(outputreg>>>(DataWidthIn+InternalFixedPoint-
DataWidthOut))<<<(DataWidthIn+InternalFixedPoint-DataWidthOut);
        end
   end

   // Making offset binary output from 2's compliment representation
   assign outputdata=({~quantout[DataWidthIn+InternalFixedPoint-
1],quantout[DataWidthIn+InternalFixedPoint-
2:DataWidthIn+InternalFixedPoint-DataWidthOut]});

   // Mult / add
   assign multOutFixed=multInputA*multInputB;
   assign multOut=multOutFixed>>>CoefFixedPoint;
   assign addOut=addInputA+addInputB;

   // Statecounter
   always @ (posedge inputclkx16 or posedge inputclk or posedge preset)
      begin
         if(preset==1)
            statecount=0;
            else if(inputclk==1)
               statecount=0;
            else
```

```
            statecount=statecount+1;
   end

// Register transfer
always @ (negedge inputclkx16 or posedge preset)
   begin
      if(preset==1) begin
         outputreg=0;
         accInput0=0;
         accInput1=0;
         accInput2=0;
         accInput3=0;
         accInput4=0;
      end
      else begin
            case(statecount)
            4'd0: begin
                  outputreg=addOut;
               end
            4'd1: begin
                  accInput4=addOut;
               end
            4'd2: begin
                  accInput4=addOut;
               end
            4'd3: begin
                  accInput3=addOut;
               end
            4'd4: begin
                  accInput3=addOut;
               end
            4'd5: begin
                  accInput3=addOut;
               end
            4'd6: begin
                  accInput2=addOut;
               end
            4'd7: begin
                  accInput2=addOut;
               end
            4'd8: begin
                  accInput1=addOut;
               end
            4'd9: begin
                  accInput1=addOut;
               end
            4'd10: begin
                  accInput1=addOut;
               end
            4'd11: begin
                  accInput0=multOut;
               end
            4'd12: begin
                  accInput0=addOut;
               end
            endcase
      end
end

// Combinatorial logic
```

```
   always @ (addOut, multOut, quantout, acc0, acc1, acc2, acc3, acc4,
statecount,
             accInput0, accInput1, accInput2, accInput3, accInput4,
inputreg)
      begin
         case(statecount)
            4'd0: begin
                     multInputA=0;
                     multInputB=0;
                     addInputA=inputreg;
                     addInputB=acc4;
                  end
            4'd1: begin
                     multInputA=B4;
                     multInputB=inputreg;
                     addInputA=multOut;
                     addInputB=acc3;
                  end
            4'd2: begin
                     multInputA=A4;
                     multInputB=quantout;
                     addInputA=multOut;
                     addInputB=accInput4;
                  end
            4'd3: begin
                     multInputA=B3;
                     multInputB=inputreg;
                     addInputA=multOut;
                     addInputB=acc2;
                  end
            4'd4: begin
                     multInputA=A3;
                     multInputB=quantout;
                     addInputA=multOut;
                     addInputB=accInput3;
                  end
            4'd5: begin
                     multInputA=G1;
                     multInputB=acc4;
                     addInputA=multOut;
                     addInputB=accInput3;
                  end
            4'd6: begin
                     multInputA=B2;
                     multInputB=inputreg;
                     addInputA=multOut;
                     addInputB=acc1;
                 end
            4'd7: begin
                     multInputA=A2;
                     multInputB=quantout;
                     addInputA=multOut;
                     addInputB=accInput2;
                  end
            4'd8: begin
                     multInputA=B1;
                     multInputB=inputreg;
                     addInputA=multOut;
                     addInputB=acc0;
                  end
            4'd9: begin
```

```
                    multInputA=A1;
                    multInputB=quantout;
                    addInputA=multOut;
                    addInputB=accInput1;
                end
            4'd10: begin
                    multInputA=G0;
                    multInputB=acc2;
                    addInputA=multOut;
                    addInputB=accInput1;
                end
            4'd11: begin
                    multInputA=B0;
                    multInputB=inputreg;
                    addInputA=0;
                    addInputB=0;
                end
            4'd12: begin
                    multInputA=A0;
                    multInputB=quantout;
                    addInputA=multOut;
                    addInputB=accInput0;
                end
            default: begin
                    multInputA=0;
                    multInputB=0;
                    addInputA=0;
                    addInputB=0;
                end
        endcase
    end

    // Accumulate and read new input data
    always @ (posedge inputclk or posedge preset)
       begin
         if (preset == 1'b1) begin
          inputreg=0;
          acc0 = 0;
          acc1 = 0;
          acc2 = 0;
          acc3 = 0;
          acc4 = 0;
       end
       else begin
          inputreg=inputdata<<<InternalFixedPoint;
          acc0 = acc0+accInput0;
          acc1 = acc1+accInput1;
          acc2 = acc2+accInput2;
          acc3 = acc3+accInput3;
          acc4 = acc4+accInput4;
       end
     end
endmodule
```

### 10.1.12 PWM

```
////////////////////////////////////////////////////////////////////////////
/////
//
// Project:       Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   PWM module
//
// Additional coment: 1-bit output, offset binary input
//
//
////////////////////////////////////////////////////////////////////////////
/////

module pwmmodule (inputdata, outputdata, inputclk, outputclk, preset);
  parameter InputWidth=4;

  input [InputWidth-1:0] inputdata;
  output outputdata;
  input   inputclk;
  input   outputclk;
  input   preset;

  reg [InputWidth-1:0] compcounter;
  reg [InputWidth-1:0] inputreg;
  reg compoutput;

  assign outputdata=compoutput;

  // Input register
  always @ (posedge inputclk or posedge preset)
    begin
      if (preset == 1'b1) begin
         inputreg=0;
      end
      else begin
         // If 2's compliment at input:
         //inputreg=({~inputdata[InputWidth-1],inputdata[InputWidth-2:0]});

         inputreg=inputdata;
      end
    end


  // Comparator sawtooth
  always @ (posedge inputclk or posedge outputclk or posedge preset)
    begin
      if (preset == 1'b1)
         compcounter=0;
      else if (inputclk == 1'b1)
         compcounter=0;
      else
         compcounter=compcounter+1;
    end

  // Compare sawtooth and input
```

```
  always @ (negedge outputclk or posedge preset)
    begin
      if (preset == 1'b1)
         compoutput=0;
      else
      if(inputreg>compcounter)
         compoutput = 1;
      else
         compoutput = 0;
  end

endmodule
```

### 10.1.13 FIR MAC interpolator, uten RAM (med registere)

```
//////////////////////////////////////////////////////////////////////////////
/////
//
// Project:             Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   MAC implementation, upsample x2 + halfband FIR
//              Uses rgisters for buffering
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
/////

module macmoduleASIC(preset, sample_in_freq, coefinput, coefaddressoutput,
mac_freq, datain, dataout);
   parameter DataWidth=17;
   parameter CoefWidth=21;
   parameter CoefAdrBitlength=5;
   parameter NoCoef=32;
   parameter FmacDivFsample=256;

   parameter InternalFixedPoint=8;
   parameter AccumulatorWidthExpand=1;
   // Requirements: mac_freq > sample_infreq*4*NoCoef+1
   // Note: Coef fixed point = CoefWidth-1

   input preset, sample_in_freq, mac_freq;
   input signed [CoefWidth-1:0] coefinput;
   input signed [DataWidth-1:0] datain;

   output [CoefAdrBitlength-1:0]coefaddressoutput;
   output signed [DataWidth-1:0] dataout;

   integer n;

   // Input buffer
   reg signed [DataWidth-1:0] inputbuffer[0:(NoCoef*2-1)];


   // MAC registers/variables
   reg signed [DataWidth-1:0] macDataIn;
   reg signed [CoefWidth-1:0] macCoefIn;

   wire signed [DataWidth+InternalFixedPoint-1:0]addIn;
   reg signed [DataWidth+InternalFixedPoint+AccumulatorWidthExpand-1:0]
accReg;

   wire signed [DataWidth+CoefWidth-1:0] multOut;
   reg signed [DataWidth-1:0] macOut;

   reg [CoefAdrBitlength-1:0]coefaddress;
   reg [(CoefAdrBitlength*2)+1:0] statecount;

   assign multOut=macDataIn*macCoefIn;
   assign addIn=multOut>>>(CoefWidth-InternalFixedPoint-1);
```

```
assign dataout=macOut;
assign coefaddressoutput=coefaddress;

// Input buffer RAM
always @ (posedge sample_in_freq or posedge preset)
   begin
      if(preset==1)
         begin
            for(n=0; n < NoCoef*2; n=n+1)
               inputbuffer[n]= 0;
         end
      else
         begin
            for(n=(NoCoef*2-1); n >0 ; n=n-1)
                  inputbuffer[n] = inputbuffer[n-1];
            inputbuffer[0] = datain;
                 end
   end

//Accumulate
always @ (posedge preset or posedge mac_freq or posedge sample_in_freq)
   begin
      if(preset==1) begin
         accReg=0;
        statecount=0; end
            else
               if (sample_in_freq==1) begin
                  accReg=0;
                           statecount=0; end
               else begin
                    accReg=accReg+addIn;
                       statecount=statecount+1;
            end
  end

// Combinatorial logic
 always @ (statecount or accReg or coefinput)
   begin
        if(statecount < NoCoef)

         begin
            coefaddress=statecount;
            macCoefIn = coefinput;
            macDataIn = inputbuffer[statecount];
         end
      else if(statecount < NoCoef*2)
         begin
            coefaddress=NoCoef*2-statecount-1;
            macCoefIn = coefinput;
            macDataIn = inputbuffer[statecount];
         end
      else
         begin
            coefaddress=0;
            macCoefIn =0;
            macDataIn=0;

         end
      if(statecount < FmacDivFsample/2)
         macOut=accReg>>>InternalFixedPoint;
```

```
         else
            macOut=inputbuffer[NoCoef-1];
      end
endmodule
```

### 10.1.14 Toppnivå DA-konverter, uten RAM

```
//////////////////////////////////////////////////////////////////////////////
//
//
// Project:       Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   Delta-Sigma DA toplevel
//                General, uses registers for input buffering
//
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////

`timescale 1ns/100ps

module dstoplevel(inputdata, outputdata, sampleclock, sampleclockOSR,
preset);
   `include "coefrom1parameters.v"
   `include "coefrom2parameters.v"
   `include "coefrom3parameters.v"

   parameter DataWidthIn=18;
   parameter  DataWidthOut=4;

   // Parameters predistortion module
   parameter PredInternalFixedPoint=0; // InternalFixedPoint
   parameter PredBetaCoef=4; // Corresponds to 1/(2^4)=0.0625

   // Parameters DS module
   parameter DSInternalWidth=24;
   parameter DSInternalFixedPoint=4; // InternalFixedPoint

   // Parameters Ram1
   parameter RAM1_AdrBits=6;
   parameter RAM1_DataWidth=24;

   // Parameters Ram2
   parameter RAM2_AdrBits=4;
   parameter RAM2_DataWidth=24;

   // Parameters Ram3
   parameter RAM3_AdrBits=4;
   parameter RAM3_DataWidth=24;

   // Inputs/outputs
   input signed [DataWidthIn-1:0] inputdata;
   output outputdata;
   input sampleclock;
   input sampleclockOSR;
   input preset;

   // General instance wires
   wire signed [DataWidthIn-1:0] macDataIn1;
   wire signed [DataWidthIn-1:0] macDataOut1;
   wire signed [DataWidthIn-1:0] macDataOut2;
   wire signed [DataWidthIn-1:0] macDataOut3;
```

```
   wire signed [DataWidthIn-1:0] cicDataOut;
   wire signed [DataWidthIn-1:0] predistortionOut;
   wire [DataWidthOut-1:0] dsDataOut;
   wire pwmDataOut;
   wire [8:0] clk_out;

   // RAM/ROM wires
   wire [CoefROM1_AdrBits-1:0] CoefROM1_address;
   wire [CoefROM2_AdrBits-1:0] CoefROM2_address;
   wire [CoefROM3_AdrBits-1:0] CoefROM3_address;

   wire [CoefROM1_CoefWidth-1:0] CoefROM1_out;
   wire [CoefROM2_CoefWidth-1:0] CoefROM2_out;
   wire [CoefROM3_CoefWidth-1:0] CoefROM3_out;

   assign macDataIn1=inputdata;
   assign outputdata=pwmDataOut;


// Instances
   coefrom1 rom1 (.CoefROM1_address(CoefROM1_address),
.CoefROM1_out(CoefROM1_out));

   coefrom2 rom2 (.CoefROM2_address(CoefROM2_address),
.CoefROM2_out(CoefROM2_out));

   coefrom3 rom3 (.CoefROM3_address(CoefROM3_address),
.CoefROM3_out(CoefROM3_out));

   clock_module klokke (.preset(preset), .sample_freq(sampleclock),
            .sample_freq512(sampleclockOSR), .fout(clk_out) );


   macmoduleASIC  #(.DataWidth(DataWidthIn),.CoefWidth(CoefROM1_CoefWidth),
.FmacDivFsample(256),
                .CoefAdrBitlength(CoefROM1_AdrBits),
.NoCoef(CoefROM1_NoCoef))
               mac1
               (.preset(preset), .sample_in_freq(clk_out[0]),
               .mac_freq(clk_out[8]), .coefinput(CoefROM1_out),
               .coefaddressoutput(CoefROM1_address),
               .datain(inputdata), .dataout(macDataOut1));

   macmoduleASIC  #(.DataWidth(DataWidthIn),.CoefWidth(CoefROM2_CoefWidth),
.FmacDivFsample(128),
                .CoefAdrBitlength(CoefROM2_AdrBits),
.NoCoef(CoefROM2_NoCoef))
               mac2
               (.preset(preset), .sample_in_freq(clk_out[1]),
               .mac_freq(clk_out[8]), .coefinput(CoefROM2_out),
               .coefaddressoutput(CoefROM2_address),
               .datain(macDataOut1), .dataout(macDataOut2));

   macmoduleASIC  #(.DataWidth(DataWidthIn),.CoefWidth(CoefROM3_CoefWidth),
.FmacDivFsample(64),
                .CoefAdrBitlength(CoefROM3_AdrBits),
.NoCoef(CoefROM3_NoCoef))
               mac3
               (.preset(preset), .sample_in_freq(clk_out[2]),
               .mac_freq(clk_out[8]), .coefinput(CoefROM3_out),
               .coefaddressoutput(CoefROM3_address),
```

```
                .datain(macDataOut2), .dataout(macDataOut3));


   cicmodule #(.DataWidth(DataWidthIn)) cic (.inputclk(clk_out[3]),
.internalclk(clk_out[8]),
                .outputclk(clk_out[5]), .preset(preset),
.datainput(macDataOut3), .dataoutput(cicDataOut) );


   predistortionmoduleA #(.DataWidth(DataWidthIn),
.InternalFixedPoint(PredInternalFixedPoint), .BetaCoef(PredBetaCoef) )
predistortion
                          (.inputdata(cicDataOut),
.outputdata(predistortionOut),
                           .inputclk(clk_out[5]),
.inputclkx16(sampleclockOSR),
                           .preset(preset));

   dsmoduleINS #(.DataWidthIn(DataWidthIn),
.DataWidthOut(DataWidthOut),.InternalWidth(DSInternalWidth),
.InternalFixedPoint(DSInternalFixedPoint)) dsIns
              (.inputdata(predistortionOut), .outputdata(dsDataOut),
.inputclk(clk_out[5]),
               .inputclkx16(sampleclockOSR), .preset(preset));


   pwmmodule #(.InputWidth(DataWidthOut)) pwm (.inputdata(dsDataOut),
.outputdata(pwmDataOut),
                       .inputclk(clk_out[5]), .outputclk(sampleclockOSR),
.preset(preset));

endmodule
```

### 10.1.15 FIR MAC interpolator med registre for buffering (ikke RAM)

```
//////////////////////////////////////////////////////////////////////////////
/////
//
// Project:             Master thesis NTNU 2006, Delta sigma DA in FPGA
//
// Autor:         Lasse Olsen
//
// Description:   MAC implementation, upsample x2 + halfband FIR
//              Uses rgisters for buffering
//
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////
/////

module macmoduleASIC(preset, sample_in_freq, coefinput, coefaddressoutput,
mac_freq, datain, dataout);
   parameter DataWidth=17;
   parameter CoefWidth=21;
   parameter CoefAdrBitlength=5;
   parameter NoCoef=32;
   parameter FmacDivFsample=256;

   parameter InternalFixedPoint=8;
   parameter AccumulatorWidthExpand=1;
   // Requirements: mac_freq > sample_infreq*4*NoCoef+1
   // Note: Coef fixed point = CoefWidth-1

   input preset, sample_in_freq, mac_freq;
   input signed [CoefWidth-1:0] coefinput;
   input signed [DataWidth-1:0] datain;

   output [CoefAdrBitlength-1:0]coefaddressoutput;
   output signed [DataWidth-1:0] dataout;

   integer n;

   // Input buffer
   reg signed [DataWidth-1:0] inputbuffer[0:(NoCoef*2-1)];


   // MAC registers/variables
   reg signed [DataWidth-1:0] macDataIn;
   reg signed [CoefWidth-1:0] macCoefIn;

   wire signed [DataWidth+InternalFixedPoint-1:0]addIn;
   reg signed [DataWidth+InternalFixedPoint+AccumulatorWidthExpand-1:0]
accReg;

   wire signed [DataWidth+CoefWidth-1:0] multOut;
   reg signed [DataWidth-1:0] macOut;

   reg [CoefAdrBitlength-1:0]coefaddress;
   reg [(CoefAdrBitlength*2)+1:0] statecount;

   assign multOut=macDataIn*macCoefIn;
```

```
assign addIn=multOut>>>(CoefWidth-InternalFixedPoint-1);

assign dataout=macOut;
assign coefaddressoutput=coefaddress;

// Input buffer RAM
always @ (posedge sample_in_freq or posedge preset)
   begin
      if(preset==1)
         begin
            for(n=0; n < NoCoef*2; n=n+1)
               inputbuffer[n]= 0;
         end
      else
         begin
            for(n=(NoCoef*2-1); n >0 ; n=n-1)
                  inputbuffer[n] = inputbuffer[n-1];
            inputbuffer[0] = datain;
                 end
   end

//Accumulate
always @ (posedge preset or posedge mac_freq or posedge sample_in_freq)
   begin
      if(preset==1) begin
         accReg=0;
        statecount=0; end
            else
               if (sample_in_freq==1) begin
                  accReg=0;
                           statecount=0; end
               else begin
                    accReg=accReg+addIn;
                       statecount=statecount+1;
            end
  end

// Combinatorial logic
 always @ (statecount or accReg or coefinput)
   begin
        if(statecount < NoCoef)

         begin
            coefaddress=statecount;
            macCoefIn = coefinput;
            macDataIn = inputbuffer[statecount];
         end
      else if(statecount < NoCoef*2)
         begin
            coefaddress=NoCoef*2-statecount-1;
            macCoefIn = coefinput;
            macDataIn = inputbuffer[statecount];
         end
      else
         begin
            coefaddress=0;
            macCoefIn =0;
            macDataIn=0;

         end
      if(statecount < FmacDivFsample/2)
```

```
            macOut=accReg>>>InternalFixedPoint;
         else
            macOut=inputbuffer[NoCoef-1];
      end
endmodule
```

## *10.2 Matlab kode*

### 10.2.1 Hovedfil, sam-simulering

```
% CO-simulation setup, DS DA-converter

DW=24;
OSR=32;
PWMFREQ=16;
TESTF=1;
FS=44.1;
cicorder=3;
div2Interpol=8;
LEVELS=16;
order=5;
opt=1;
gain=1.6;
%sine=round((((2^22)-1)*(rand(1, 1024)-(1/2))));
l=256;
%l=FS*16;
%signal=floor(2^(DW-1)*(8./10)*cos(2*pi*(TESTF/FS)*(1:l)));
signalUpsampled=floor(2^(DW-
1)*(9/10)*cos(2*pi*(TESTF/(FS*OSR))*(1:l*OSR)));
signalUpsampled=signalUpsampled*LEVELS/(2^(DW-1));
%signal=floor(2^(DW-1)*(8/10)*(2*rand(1, l)-1));

signal=2^(DW-1)*(8/10)*cos(2*pi*(TESTF/FS)*(1:l));

%signal=ceil(signal);

% 18-bit signal with 16-bit quant
%signal=floor(signal/4)*4;

signal=floor(signal);

sine1=2^(DW-1)*(8/10)*cos(2*pi*(1/44)*(1:l))/5;
sine2=2^(DW-1)*(8/10)*cos(2*pi*(3/44)*(1:l))/5;
sine3=2^(DW-1)*(8/10)*cos(2*pi*(9/44)*(1:l))/5;
sine4=2^(DW-1)*(8/10)*cos(2*pi*(14/44)*(1:l))/5;
sine5=2^(DW-1)*(8/10)*cos(2*pi*(19/44)*(1:l))/5;
%signal=floor(sine1+sine2+sine3+sine4+sine5);

%signal=floor(2^(DW-1)*(9/10)*(2*rand(1, l)-1));
%signal=2^(DW-1)*wavread('lyd', [7091000 7095000]);

%signal=2^(DW-1)*wavread('lyd3', [10049000 10052000]);
%signal=signal(:, 1);
%signal=signal';

genverilogdat2(signal, DW, 'C:\Documents and Settings\Lasse Hjem\Mine
dokumenter\Diplom\DS_verilog\sine24bit.txt');

%signalUpsampled=signalUpsampled*LEVELS/(2^(DW-1));
signal=signal*LEVELS/(2^(DW-1));
```

```
sigIn=signal;

close all;


%---- Axis properties ---
ymin=-270;
ymax=10;

plot(linspace(0,1,length(sigIn)),(dbv(fft(hann(length(sigIn)).*sigIn/(4*len
gth(sigIn))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('Inngangssignal');

%----- Interpol 1
fprintf(1,'Upsample x 2\n');
sigIn=upsample(sigIn, 2);
b=half1;
sigIn=filter(b,1,sigIn);

% ------ Verilog MAC1
fclose('all');
a=fscanf(fopen('C:\Documents and Settings\Lasse Hjem\Mine
dokumenter\Diplom\DS_verilog\macout1.txt'),'%d');
a=a*LEVELS/2^(DW-1);

subplot(211);
plot(linspace(0,1,length(sigIn)),(dbv(fft(hann(length(sigIn)).*sigIn/(4*len
gth(sigIn))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('FIR 1 utgangssignal, Matlab-modell');

subplot(212);
a=a';
plot(linspace(0,1,length(a)),(dbv(fft(hann(length(a)).*a/(4*length(a))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('FIR 1 utgangssignal, Verilog-implementering');


%----- Interpol 2
fprintf(1,'Upsample x 4\n');
sigIn=upsample(sigIn, 2);
b=half2;
sigIn=filter(b,1,sigIn);

%------ Verilog MAC 2
fclose('all');
a=fscanf(fopen('C:\Documents and Settings\Lasse Hjem\Mine
dokumenter\Diplom\DS_verilog\macout2.txt'),'%d');
a=a*LEVELS/2^(DW-1);
```

```
subplot(211);
plot(linspace(0,1,length(sigIn)),(dbv(fft(hann(length(sigIn)).*sigIn/(4*len
gth(sigIn))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('FIR 2 utgangssignal, Matlab-modell');

subplot(212);
a=a';
plot(linspace(0,1,length(a)),(dbv(fft(hann(length(a)).*a/(4*length(a))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('FIR 2 utgangssignal, Verilog-implementering');

%----- Interpol 3
fprintf(1,'Upsample x 8\n');
sigIn=upsample(sigIn, 2);
b=half3;
sigIn=filter(b,1,sigIn);

%------ Verilog MAC3
fclose('all');
a=fscanf(fopen('C:\Documents and Settings\Lasse Hjem\Mine
dokumenter\Diplom\DS_verilog\macout3.txt'),'%d');
a=a*LEVELS/2^(DW-1);

subplot(211);
plot(linspace(0,1,length(sigIn)),(dbv(fft(hann(length(sigIn)).*sigIn/(4*len
gth(sigIn))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('FIR 3 utgangssignal, Matlab-modell');

subplot(212);
a=a';
plot(linspace(0,1,length(a)),(dbv(fft(hann(length(a)).*a/(4*length(a))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('FIR 3 utgangssignal, Verilog-implementering');

%----- CIC ------
fprintf(1,'CIC --> Upsample x %d\n',OSR);
lengthIn=length(sigIn);
holdRate=OSR/div2Interpol;

for i=1:cicorder
    sigIn=filter([1 -1],1,sigIn);
end;
sigIn=upsample(sigIn, OSR/div2Interpol);
for i=1:cicorder
   sigIn=filter(1,[1 -1],sigIn);
end;
```

```
sigIn = sigIn/(OSR/div2Interpol)^(cicorder-1);

%--- Plot diference filter/no filter

sigInNoFilter=upsample(signal, OSR)*OSR;

regsizefiltlin=abs(max(sigIn))/abs(max(signal))
regsizefilt=dbv(max(sigIn)/max(signal))

regsizeNofiltlin=abs(max(sigInNoFilter))/abs(max(signal))
regsizefilt=dbv(max(sigInNoFilter)/max(signal))

%figure;
%subplot(211);
%plot(linspace(0,1,length(sigIn)),(dbv(fft(hann(length(sigIn)).*sigIn/(4*le
ngth(sigIn))))));
%subplot(212);
%plot(linspace(0,1,length(sigInNoFilter)),(dbv(fft(hann(length(sigInNoFilte
r)).*sigInNoFilter/(4*length(sigInNoFilter))))),'r');

%rms(sigIn);
%rms(sigInNoFilter);

%axis([0 0.5 ymin ymax]);
%grid on;
%xlabel('Normalisert frekvens (1\rightarrow f_s)');
%ylabel('dB');
%title('CIC-filter utgangssignal, Verilog-implementering');


%  ------ Verilog cic
fclose('all');
a=fscanf(fopen('C:\Documents and Settings\Lasse Hjem\Mine
dokumenter\Diplom\DS_verilog\cicoutput.txt'),'%d');
a=a*LEVELS/2^(DW-1);

subplot(211);
plot(linspace(0,1,length(sigIn)),(dbv(fft(hann(length(sigIn)).*sigIn/(4*len
gth(sigIn))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('CIC-filter utgangssignal, Matlab-modell');

subplot(212);
a=a';
plot(linspace(0,1,length(a)),(dbv(fft(hann(length(a)).*a/(4*length(a))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('CIC-filter utgangssignal, Verilog-implementering');

%sigIn=signalUpsampled;

%powSigIn=sum(sigIn.^2)
%powSigInUpsampled=sum(signalUpsampled.^2)


%----- Predistortion -----
```

```
sigIn=makenatural5(sigIn,LEVELS, 0.0625);

%Verilog predistortion
fclose('all');
a=fscanf(fopen('C:\Documents and Settings\Lasse Hjem\Mine
dokumenter\Diplom\DS_verilog\predistout.txt'),'%d');
a=a*LEVELS/2^(DW-1);

subplot(211);
plot(linspace(0,1,length(sigIn)),(dbv(fft(hann(length(sigIn)).*sigIn/(4*len
gth(sigIn))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('Predistortion utgangssignal, Matlab-modell');

subplot(212);
a=a';
plot(linspace(0,1,length(a)),(dbv(fft(hann(length(a)).*a/(4*length(a))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('Predistortion utgangssignal, Verilog-implementering');

%----- DS -----
H = synthesizeNTF(order,OSR,opt,gain);
sigIn=simulateDSMcifb5(sigIn, H,LEVELS);

%[a, g, b, c]=realizeNTF(H, 'CIFB');
%ABCD=stuffABCD(a, g, b, c, 'CIFB');
%sigIn=simulateDSM(sigIn, ABCD, LEVELS);

% Verilog DS
fclose('all');
a=fscanf(fopen('C:\Documents and Settings\Lasse Hjem\Mine
dokumenter\Diplom\DS_verilog\dsout.txt'),'%d');
a=(a-LEVELS/2)*2;

subplot(211);
plot(linspace(0,1,length(sigIn)),(dbv(fft(hann(length(sigIn)).*sigIn/(4*len
gth(sigIn))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('DS-modulator utgangssignal, Matlab-modell');

subplot(212);
a=a';
plot(linspace(0,1,length(a)),(dbv(fft(hann(length(a)).*a/(4*length(a))))));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('DS-modulator utgangssignal, Verilog-implementering');

% ------ Plot PWM out

sigIn=(sigIn+LEVELS-1)/(2*LEVELS);
```

```
sigIn=makeDoublePWM(sigIn, LEVELS);

spectM=fft(hann(length(sigIn)).*sigIn*4/(length(sigIn)));
spectM(1:2)=spectM(3);

snrM=calculateSNR(spectM(1:(round((length(spectM)*(20/22))/((OSR*2*PWMFREQ)
))))), ceil(length(spectM)*(TESTF/((OSR*FS*PWMFREQ)))),1)

subplot(211);
plot(linspace(0,LEVELS,length(spectM)),dbv(spectM));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('PWM utgangssignal, Matlab-modell');

fclose('all');
a=fscanf(fopen('C:\Documents and Settings\Lasse Hjem\Mine
dokumenter\Diplom\DS_verilog\pwmout.txt'),'%d');

a=a';
spectV=fft(hann(length(a)).*a*8/(length(a)));
spectV(1:2)=spectV(3);
snrV=calculateSNR(spectV(1:(round((length(spectV)*(20/22))/((OSR*2*PWMFREQ)
))))), ceil(length(spectV)*(TESTF/((OSR*FS*PWMFREQ)))),1)

subplot(212);
a=a';
a=a-0.5;
plot(linspace(0,LEVELS,length(spectV)),dbv(spectV));
axis([0 0.5 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('PWM utgangssignal, Verilog-implementering');

subplot(211);

axis([0 0.5/32 ymin ymax]);
subplot(212);
axis([0 0.5/32 ymin ymax]);

figure;
%---- LOG graph
subplot(211);
semilogx(linspace(0,LEVELS,length(spectM)),spectM);
axis([0 0.5/32 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('PWM utgangssignal, Matlab-modell');

subplot(212);
semilogx(linspace(0,LEVELS,length(spectV)),spectV);
axis([0 0.5/32 ymin ymax]);
grid on;
xlabel('Normalisert frekvens (1\rightarrow f_s)');
ylabel('dB');
title('PWM utgangssignal, Verilog-implementering');
fclose('all');
```

### 10.2.2 DS-modulator, CIFB 5. ordens hardware-nær modell

```
function [v] = simulateDSMcifb5(u, ntf, nlev)
% Hardware-close DS-modulator, CIFB 5. order structure

intvars(1:6)=0;
y=0;
[a, g, b, c]=realizeNTF(ntf,'CIFB');

out=0;
lengde=length(u);
outvect(1:lengde)=0;
intreg(1:lengde,1:6)=0;
yreg(1:lengde)=0;
for(n=1:lengde)
   [y intvars]=cifb5(u(n),out,intvars, a, g, b, c);
   %intreg(n,1:6)=ceil(intvars*(2^23/nlev));
   intreg(n,1:6)=intvars;

   y=y+1;
   if(y >= nlev)
      out=nlev-1;
   elseif(y<-(nlev-1))
         out=-(nlev-1);
      else
        out=floor(((y-1)/2))*2+1;
      end
   yreg(n)=y;
   outvect(n)=out;
end;

preqsize=dbv(max(yreg(1:end))/nlev)
size1=dbv(max(intreg(1:end, 2))/nlev)
size2=dbv(max(intreg(1:end, 3))/nlev)
size3=dbv(max(intreg(1:end, 4))/nlev)
size4=dbv(max(intreg(1:end, 5))/nlev)
size5=dbv(max(intreg(1:end, 6))/nlev)

v=outvect;


function [y intstate] = cifb5(datain, fbin, intstate, a, g, b, c)

%intstates
%1: prev data in
%2: xo1
%3: xo2
%4: xo3
%5: xo4
%6: x05


xo(5)=intstate(6)+intstate(1)*b(5)+intstate(5)*c(4)-fbin*a(5);
xo(4)=intstate(5)+intstate(1)*b(4)+intstate(4)*c(3)-fbin*a(4)-
g(2)*intstate(6);
xo(3)=intstate(4)+intstate(1)*b(3)+intstate(3)*c(2)-fbin*a(3);
xo(2)=intstate(3)+intstate(1)*b(2)+intstate(2)*c(1)-fbin*a(2)-
g(1)*intstate(4);
xo(1)=intstate(2)+intstate(1)*b(1)-fbin*a(1);
```

```
y=datain*b(6)+xo(5)*c(5);

intstate(1)=datain;
intstate(2)=xo(1);
intstate(3)=xo(2);
intstate(4)=xo(3);
intstate(5)=xo(4);
intstate(6)=xo(5);
```

### 10.2.3 Funksjon for skriving av data til fil for Modelsim

```
function genverilogdat2(inputdata, datawidth, filename)
%genverilogdat2(coefwidth, b,filename)
fid=fopen(filename,'w');

for i=1:length(inputdata)
   fprintf(fid,dec2hex2s(inputdata(i),datawidth));
   fprintf(fid,'\n');
end

fclose(fid);
```

### 10.2.4 Konfigurering FIR 1

```
function [qcoef fixedcoef] = half1
%[qcoef fixedcoef] = half1

%
% M-File generated by MATLAB(R) 7.1 and the Signal Processing Toolbox 6.4.
%
% Generated on: 15-Feb-2006 13:41:58
%

% Equiripple Lowpass filter designed using the FIRPM function.

% All frequency values are in Hz.
% Data to keep BEFORE filter B is halfed

Fs = 44100;  % Sampling Frequency
Coeflength = 17; % Length abs(coef), length included sign = length+1
N     = 63;     % Order
Fpass = 20000;  % Passband Frequency
Fstop = 22050;  % Stopband Frequency
Wpass = 1;      % Passband Weight
Wstop = 1;      % Stopband Weight
dens  = 20;     % Density Factor

% Calculate the coefficients using the FIRPM function.
b  = firpm(N, [0 Fpass Fstop Fs/2]/(Fs/2), [1 1 0 0], [Wpass Wstop], ...
           {dens});
%Hd = dfilt.dffir(b);
%figure(1); clf;
%freqz(makehalf(b));
%figure(2); clf;

b=quant(makehalf(b),2^Coeflength);
resp=freqz(b);
%freqz(b);
bot=min(dbv(resp(1:round((end*(Fpass/(2*Fs)))))));
top=max(dbv(resp(1:round((end*(Fpass/(2*Fs)))))));

stopband=dbv(max((resp(round(end*(2*Fs-Fpass)/(2*Fs)):end))))-6
stopband=dbv(rms((resp(round(end*(2*Fs-Fpass)/(2*Fs)):end))))-6

ripple=top-bot
qcoef=b;
```

```
fixedcoef=b*2^Coeflength;
genverilogdat(Coeflength+1,fixedcoef,'coefrom1.v',
'coefrom1parameters.v',1);
fprintf(1, '\nMax passband ripple: %1.6fdB\n\n',ripple);
```

### 10.2.5 Konfigurering FIR 2

```
function [qcoef fixedcoef] = half1
%TULL Returns a discrete-time filter object.

%
% M-File generated by MATLAB(R) 7.1 and the Signal Processing Toolbox 6.4.
%
% Generated on: 15-Feb-2006 13:41:58
%

% Equiripple Lowpass filter designed using the FIRPM function.

% All frequency values are in Hz.
% Data to keep BEFORE filter B is halfed

Fs = 88200;  % Sampling Frequency
Coeflength = 14;% Length abs(coef), length included sign = length+1
N     = 15;     % Order
Fpass = 20000;  % Passband Frequency
Fstop = 44100;  % Stopband Frequency
Wpass = 1;      % Passband Weight
Wstop = 1;      % Stopband Weight
dens  = 20;     % Density Factor

% Calculate the coefficients using the FIRPM function.
b  = firpm(N, [0 Fpass Fstop Fs/2]/(Fs/2), [1 1 0 0], [Wpass Wstop], ...
           {dens});
%Hd = dfilt.dffir(b);
%figure(1); clf;
%freqz(makehalf(b));
%figure(2); clf;

b=quant(makehalf(b),2^Coeflength);
resp=freqz(b);
%freqz(b);
bot=min(dbv(resp(1:round((end*(Fpass/(2*Fs)))))));
top=max(dbv(resp(1:round((end*(Fpass/(2*Fs)))))));

stopband=dbv(max((resp(round(end*(2*Fs-Fpass)/(2*Fs)):end))))-6
stopband=dbv(rms((resp(round(end*(2*Fs-Fpass)/(2*Fs)):end))))-6

ripple=top-bot
qcoef=b;
fixedcoef=b*2^Coeflength;
genverilogdat(Coeflength+1,fixedcoef,'coefrom2.v',
'coefrom2parameters.v',2);
fprintf(1, '\nMax passband ripple: %1.6fdB\n\n',ripple);

% [EOF]
```

### 10.2.6 Konfigurering FIR 3

```
function [qcoef fixedcoef] = half1
%TULL Returns a discrete-time filter object.

%
% M-File generated by MATLAB(R) 7.1 and the Signal Processing Toolbox 6.4.
%
% Generated on: 15-Feb-2006 13:41:58
%

% Equiripple Lowpass filter designed using the FIRPM function.

% All frequency values are in Hz.
% Data to keep BEFORE filter B is halfed

Fs = 176400;  % Sampling Frequency
Coeflength = 9;  % Length abs(coef), length included sign = length+1
N     = 7;     % Order
Fpass = 20000;  % Passband Frequency
Fstop = 88200;  % Stopband Frequency
Wpass = 1;      % Passband Weight
Wstop = 1;      % Stopband Weight
dens  = 20;     % Density Factor

% Calculate the coefficients using the FIRPM function.
b  = firpm(N, [0 Fpass Fstop Fs/2]/(Fs/2), [1 1 0 0], [Wpass Wstop], ...
           {dens});
%Hd = dfilt.dffir(b);
%figure(1); clf;
%freqz(makehalf(b));
%figure(2); clf;

b=quant(makehalf(b),2^Coeflength);
resp=freqz(b);
%freqz(b);

bot=min(dbv(resp(1:round((end*(Fpass/(2*Fs)))))));
top=max(dbv(resp(1:round((end*(Fpass/(2*Fs)))))));

stopband=dbv(max((resp(round(end*(2*Fs-Fpass)/(2*Fs)):end))))-6
stopband=dbv(rms((resp(round(end*(2*Fs-Fpass)/(2*Fs)):end))))-6

ripple=top-bot
qcoef=b;
fixedcoef=b*2^Coeflength;
genverilogdat(Coeflength+1,fixedcoef,'coefrom3.v',
'coefrom3parameters.v',3);
fprintf(1, '\nMax passband ripple: %1.6fdB\n\n',ripple);
% [EOF]
```

### 10.2.7 Funksjon for generering av verilog-kode og parametere fra FIR konfiigurasjon

```
function genverilogdat(coefwidth, b,modulefilename, parameterfilename,
romno)
%genverilogdat(coefwidth, b,filename)
%   Detailed explanation goes here
n=length(b)/2;
nocoef = n/2;
abl=ceil(log(nocoef)/log(2));

fid=fopen(modulefilename,'w');
fid2=fopen(parameterfilename,'w');

fprintf(fid2,'parameter CoefROM%d_CoefWidth=%d;\n', romno, coefwidth);
fprintf(fid2,'parameter CoefROM%d_NoCoef=%d;\n', romno, nocoef);
fprintf(fid2,'parameter CoefROM%d_AdrBits=%d;\n\n', romno, abl);

fprintf(fid,'module coefrom%d(CoefROM%d_address, CoefROM%d_out);\n\n',
romno, romno, romno);

fprintf(fid,'   input [%d:0] CoefROM%d_address;\n', (abl-1), romno);
fprintf(fid,'   output signed [%d:0] CoefROM%d_out;\n\n', (coefwidth-1),
romno);

fprintf(fid,'   reg signed [%d:0] CoefROM%d_outreg;\n', (coefwidth-1),
romno);

fprintf(fid,'   assign CoefROM%d_out=CoefROM%d_outreg;\n\n', romno, romno);

fprintf(fid,'\n   always @ (CoefROM%d_address)\n   begin\n
case(CoefROM%d_address)\n',romno, romno);
for i=1:2:n
   if(b(i)< 0)
      fprintf(fid,'         %d''d%d: CoefROM%d_outreg=-%d''sd%d;\n', abl,
(i-1)/2, romno, coefwidth,abs(b(i)));
   else
      fprintf(fid,'         %d''d%d: CoefROM%d_outreg=%d''sd%d;\n', abl,
(i-1)/2, romno, coefwidth,b(i));
   end
end

fprintf(fid,'      endcase\n   end');
fprintf(fid,'\nendmodule\n');


fclose(fid);
fclose(fid2);
```

### 10.2.8 Predistortion algoritme A

```
function ret = makenatural5( input, levels, coef)
%
var(1:length(input))=0;
saw=linspace(-1, 1, levels+1);


%N=levels-1;
N=levels;


input=input/N; % Normalize to 1

for(i=1:length(input)-3)
   slope=(1/2)*(input(i+2)-input(i+1));
   inputsum1=input(i+1)+input(i+2);
   yz=(1/2)*inputsum1;
   %tx=yz/(1-slope);


tx=yz*(1+slope+slope^2+slope^3+slope^4+slope^5+slope^6+slope^7+slope^8+slop
e^9)
   ;%+slope^10+slope^11+slope^12+slope^13);
   yx=tx;
   var(i)=N*(yx+(coef*(inputsum1-input(i)-input(i+3))*(1-tx^2)));

end;
ret=var;
```

### 10.2.9 Predistortion-algoritme B

```
function ret = makenatural6( input, levels, coef, alpha, gamma)
%
var(1:length(input))=0;
%N=levels-1;
N=levels;
input=input/N; % Normalize to 1

for(i=1:length(input)-3)
   sum1=input(i+1)+input(i+2);
   yz=sum1/2;
   y0=yz+coef*(sum1-input(i)-input(i+3));

if(y0>0)
   slope=(input(i+2)-y0);
   tx=y0/(1-slope);
   yx=tx;
   pa=y0-slope;
   da=pa-input(i+1);
   pd=y0+slope*3;
   dd=pd-input(i+3);
   var(i)=yx+(alpha*(da+(gamma*dd))*tx*(1-tx));;
else
   slope=y0-input(i+1);
   tx=y0/(1-slope);
   yx=tx;
   pb=y0+slope;
   db=pb-input(i+2);
```

```
   pc=y0-3*slope;;
   dc=pc-input(i);
   var(i)=yx-(alpha*(db+(gamma*dc))*tx*(1+tx));
end

end
ret=var*N;
```