

# Journal 1

Indlejret Signal Behandling

Gruppe 1

| Navn                             | Studienummer | Retning |
|----------------------------------|--------------|---------|
| <b>Mathias Ørnstrup Hvidberg</b> | 201905706    | E       |
| <b>Niels Højrup Pedersen</b>     | 201604812    | E       |
| <b>Jakob Saugbjerg Lange</b>     | 201907544    | E       |

## Indhold

|   |    |
|---|----|
| Introduktion .....                                    | 2  |
| Opgave A - myMultiply .....                           | 2  |
| Opgave B - Floating and fixed-point behaviour .....   | 6  |
| Simuleret ADC kvantisering .....                      | 7  |
| Simuleret kvantisering af filter-koefficienter .....  | 8  |
| Simuleret kvantisering af produkterne i filteret..... | 10 |
| Opgave C - Implementering af notch-filter .....       | 13 |

## Introduktion

I denne journal vil der blive udviklet en assembly-funktion, som kan multiplicere to tal sammen i formatet Q1.15. Efterfølgende vil der blive undersøgt hvordan og hvorledes floating og fixpoint opfører sig, samt hvordan kvantificering har betydning for et udleveret talespor. Til sidst vil der blive implementeret et notch-filter på BlackFin-processoren. Notch-filteret ville fjerne en bestemt frekvens, i det udleveret talespor. Resultat af dette notch-filter, vil blive eftervist med billeder af et oscilloskop.

## Opgave A - myMultiply

I denne opgave udvikles en assembly-funktion, som tager to shorts og ganger med hinanden. Funktionen 'myMultiply' tager to shorts (16 bit) i fixed point formatet Q1.15, ganger de to tal med hinanden og give resultatet i Q1.15 format.

På Figur 1 ses prototypen for myMultiply-funktionen. Funktionen er deklareret som værende en extern funktion, hvilket fortæller til compileren, at funktionen er deklareret i en anden fil. Dette er grundet at myMultiply skrives i assembly, og derfor ikke har en tilhørende .h-fil der kan inkluderes.

```
extern short myMultiply(short, short );
```

Figur 1 - myMultiply prototype

På Figur 2 ses implementeringen af myMultiply i assembly. De to parametre bliver lagt over i henholdsvis R0 og R1. Da de to input er 16 bit, er det kun R0.L og R1.L som er interessant. Dette er grundet at arkitekturen i Blackfin (BF) opdeler Rx i to 16 bit registre, henholdsvis MSB: Rx.H og LSB: Rx.L, som til sammen udgør et samlet register på 32 bit.

Resultatet af multiplikationen bliver gemt i R2. Fordi formatet er Q1.15, bliver R2 bit shifted ned 16 pladser, og gemmes til sidst i R0. Det resulterer i at R0 bliver lig med R2.H.

Efterfølgende bliver der returneret tilbage til main med assembly-instruktionen RTS (ReTurn from Subroutine).

```
.GLOBAL _myMultiply;

_myMultiply:

    // R0 lower ganges med R1 lower
    R2 = R0.L * R1.L;

    // Bit shifter R2 16 gange, grundet R0 = 32 bit.
    // Gemmer resultatet i R0
    // R2.H bliver shiftet ned i R0.L
    R0 = R2 >>> 16;

    // Return
    RTS;

_myMultiply.end:
```

Figur 2 - myMultiply assembly funktion

På Figur 3 ses testprogrammet som bliver brugt til at teste funktionalitet af myMultiply. Der bliver deklareret to shorts: asm1 og asm2. De to har værdien "0x4000" i Q1.15 format. Det har værdien

0.5 i decimal. Derudover bliver der lavet en variabel til at gemme resultatet. Resultatet af myMultiply-funktionen bliver til sidste i programmet udskrevet i terminalen i hex-format.

```
#include <stdio.h>

/* Managed drivers and/or services include */
#include "system/adi_initialize.h"

extern short myMultiply( short, short );

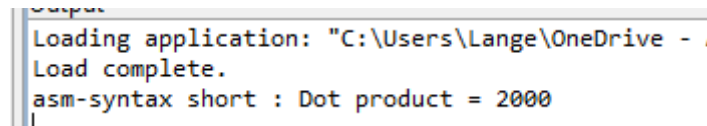
int main(void)
{
    // 0x4000 = 0.5 i Q.15
    short asm1 = 0x4000;
    short asm2 = 0x4000;
    short resultAsm = 0;

    /* Initialize managed drivers and/or services */
    adi_initComponents();

    while (1)
    {
        resultAsm = myMultiply(asm1, asm2);
        printf( "asm-syntax short : Dot product = %x\n", resultAsm );
    }
}
```

Figur 3 - Testprogram

På Figur 4 ses resultatet af testprogrammet, hvor de to hex-værdier 0x4000 bliver ganget sammen. Resultatet er udskrevet som hex, og har værdien 0x2000.



```
Loading application: "C:\Users\Lange\OneDrive - ...",
Load complete.
asm-syntax short : Dot product = 2000
```

Figur 4 - Udskrift fra terminal

Ved at bruge "myh2d" funktionen i Matlab, kan det eftervises at 0x4000 i formatet Q1.15, er lig med 0.5 decimal. På samme måde kan det eftervises, at 0x2000 i Q1.15 er lig med 0.2500 decimal som også er det der er udregnet på BF, se Figur 5.

```
>> myh2d('0x4000',1,15)

ans =

    0.5000

>> myh2d('0x2000',1,15)

ans =

    0.2500
```

Figur 5 - Eftervisning af at myMultiply regner korrekt.

Ved at udnytte debuggeren i CorssCore, kan de forskellige registre undersøges. På Figur 6 ses de forskellige registre som funktionen myMultiply anvender. På figuren ses det at R0.L og R1.L har læst inputværdierne 0x4000, hvilket stemmer overens med testprogrammet.

| Data Register File* X |          |
|-----------------------|----------|
| Name                  | Value    |
| 1010<br>0101 R0       | 00004000 |
| 1010<br>0101 R0.L     | 4000     |
| 1010<br>0101 R0.H     | 0000     |
| 1010<br>0101 R0.B     | 00       |
| 1010<br>0101 R1       | 00004000 |
| 1010<br>0101 R1.L     | 4000     |
| 1010<br>0101 R1.H     | 0000     |
| 1010<br>0101 R1.B     | 00       |
| 1010<br>0101 R2       | 00000026 |
| 1010<br>0101 R2.L     | 0026     |
| 1010<br>0101 R2.H     | 0000     |
| 1010<br>0101 R2.B     | 26       |

Figur 6 - Overview af registre fra R0-R2

På Figur 7 ses det hvordan R0 bliver lig med det som står i R2.H. R2 bliver bitshiftet ned 16 pladser, for at sikre at der bliver lagt 0 ind på de resterende pladser. På den måde bliver R0.H = 0x000, og påvirker derfor ikke det samlede resultatet.

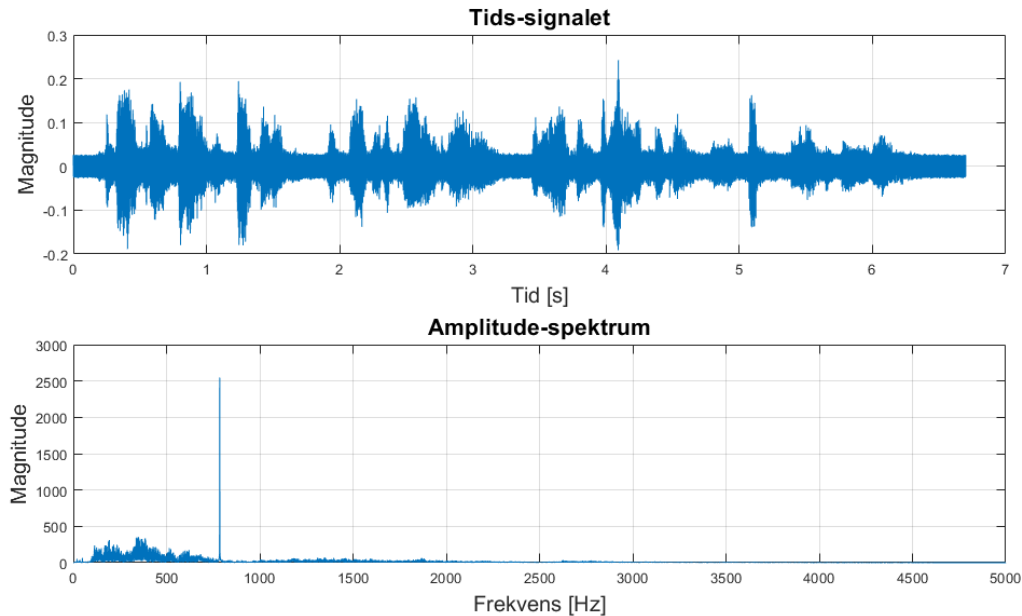
| Data Register File* X             |          |
|-----------------------------------|----------|
| Name                              | Value    |
| ✓ <small>1010<br/>0101</small> R0 | 00002000 |
| <small>1010<br/>0101</small> R0.L | 2000     |
| <small>1010<br/>0101</small> R0.H | 0000     |
| <small>1010<br/>0101</small> R0.B | 00       |
| ✓ <small>1010<br/>0101</small> R1 | 00004000 |
| <small>1010<br/>0101</small> R1.L | 4000     |
| <small>1010<br/>0101</small> R1.H | 0000     |
| <small>1010<br/>0101</small> R1.B | 00       |
| ✓ <small>1010<br/>0101</small> R2 | 20000000 |
| <small>1010<br/>0101</small> R2.L | 0000     |
| <small>1010<br/>0101</small> R2.H | 2000     |
| <small>1010<br/>0101</small> R2.B | 00       |

Figur 7 - R2 bliver bitshiftet.  $R0 = R2 \ggg 16$ .

Det kan konkluderes at assembly-funktionen myMultiply kan udregne produktet af to shorts i formatet Q1.15, og returnere et korrekt resultat i formatet Q1.15

## Opgave B - Floating and fixed-point behaviour

I denne opgave undersøges effekten af den kvantisering der finder sted, når værdier går fra floating point til fixed point. I matlab er lyd-filen "ikh\_plus\_tone\_48000hz.wav" samplet. Lydfilen plottes i tidsdomænet og i frekvensdomænet, se Figur 8.

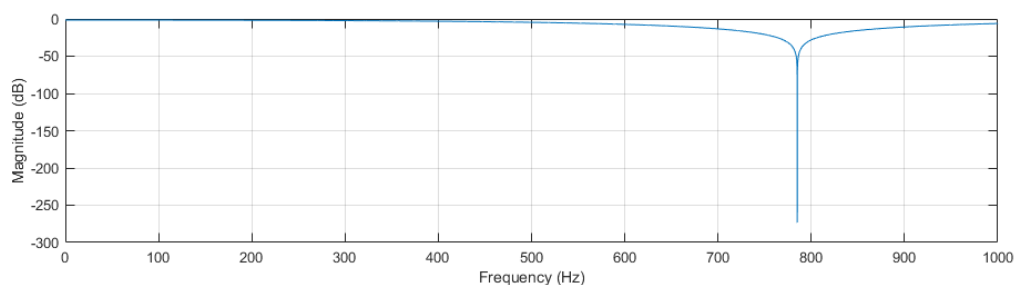


Figur 8 - Lyd-filen plottet i tidsdomænet og i frekvensdomænet

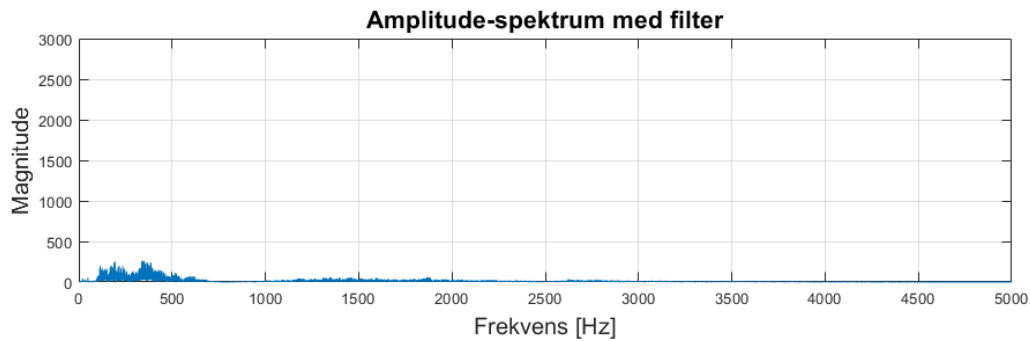
Der designes hertil et notch-filter til fjernelse af den høje sinus-tone ved frekvensen:

$$f_c = 785\text{Hz}$$

Filteret plottes også i matlab, se Figur 9 og påføres lydsignalet, hvormed signalet filtreres som forventet, hvilket kan ses på det filtrerede signals amplitude-spektre, se Figur 10.



Figur 9 - Notch-filter med cut-frekvens på 785Hz



Figur 10 - Amplitude-spektre af det filtrerede lyd-signal

### Simuleret ADC kvantisering

Hvert sample har en meget høj opløsning, men hvis disse omdannes til fixed-point-værdier i Q1.15, Q1.8 eller sågar helt ned til Q1.3 resulterer dette i en kvantisering af signalet. I Q1.15 repræsenteres hvert sample kun af 16 bit, i Q1.7 af 8 bit og i Q1.3 af kun 4 bit.

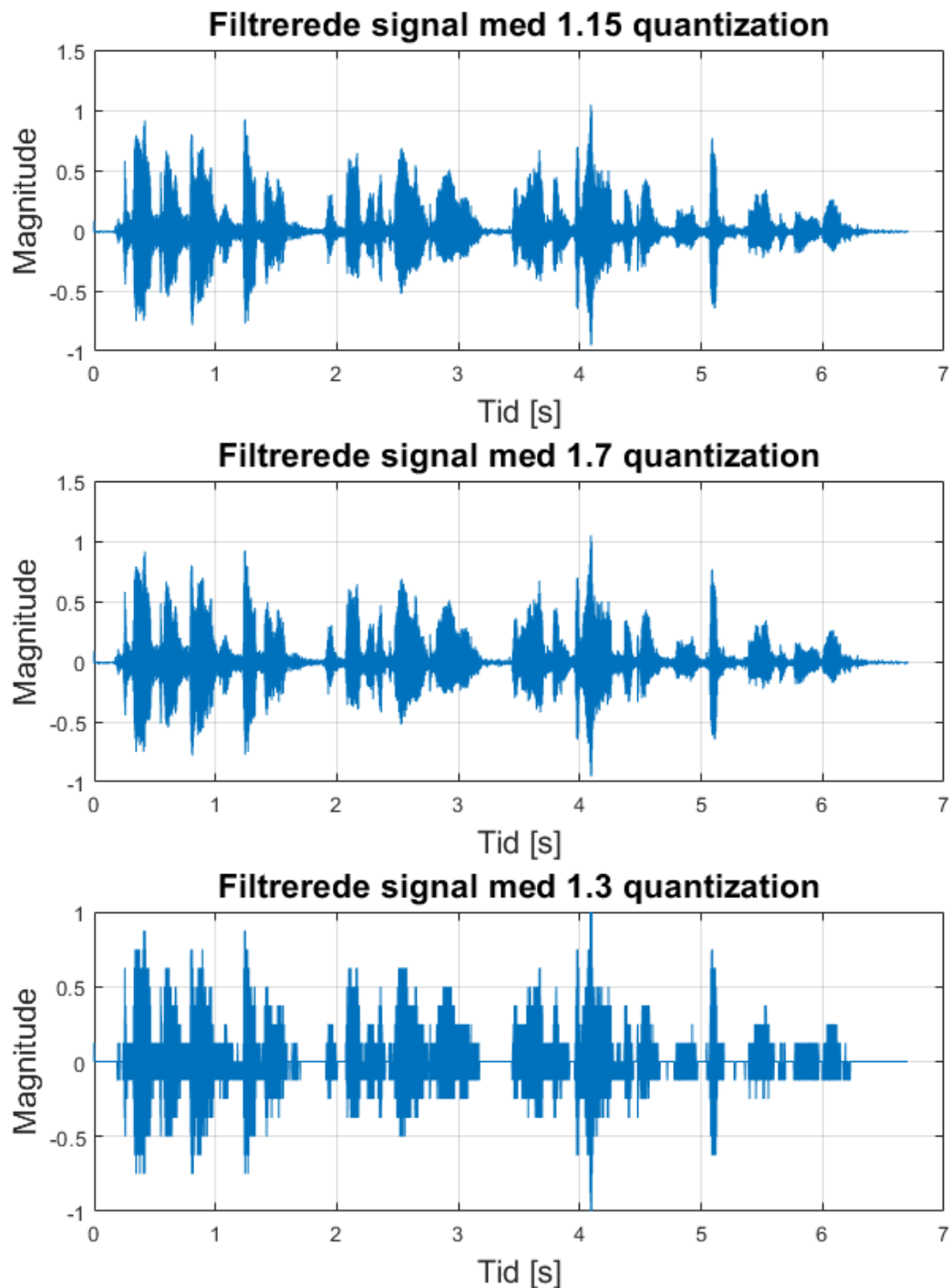
For at omdanne signalet til 16 bit, implementeres matlab-funktionen i Figur 11. For at denne funktion virker efter hensigten, skaleres signalet til værdier mellem -1 og 1.

```
function k = quantize(x, bits)
    k = round(x*2^(bits-1))*2^-(bits-1);
end
```

Figur 11 - Matlab-funktionen quantize, til kvantisering af data

På Figur 12 ses det filtrerede lyd-signal, hvor det er kvantiseret til de forskellige fixed point formater. Visuelt ses der ikke den store forskel fra det originale lyd-signal til Q1.15 og Q1.7, men lytter man til signalerne, ved brug af matlab-funktionen sound(), høre man at Q1.15-signalet ind imellem klipper lidt i signalet. Ved signalet kvantiseret til Q1.7 format, altså 8 bit, er der en tydelig konstant hvid støj, som fylder meget i forhold til det originale signal. Dette kan skyldes at signalet pga. kvantiseringen bliver mere kantet og dermed får nye frekvenser over hele spektret. Et firkant-signal indeholder fx alle frekvenser, hvormed et mere kantet signal vil resultere i mere hvid støj. I Q1.3-signalet er den hvide støj næsten overdøvende i forhold til original-signalet, men her er det også tydeligt at se på plottet, hvor signalet praktisk talt er firkantet.

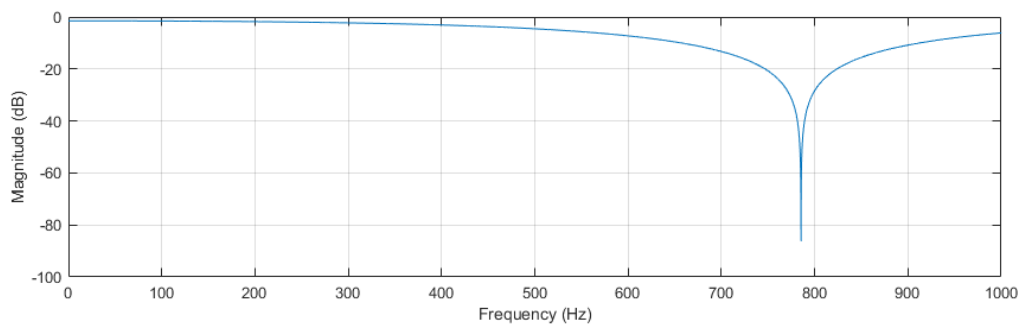




Figur 12 - Lydsignalet kvantiseret til Q1.15, Q1.7 og Q1.3.

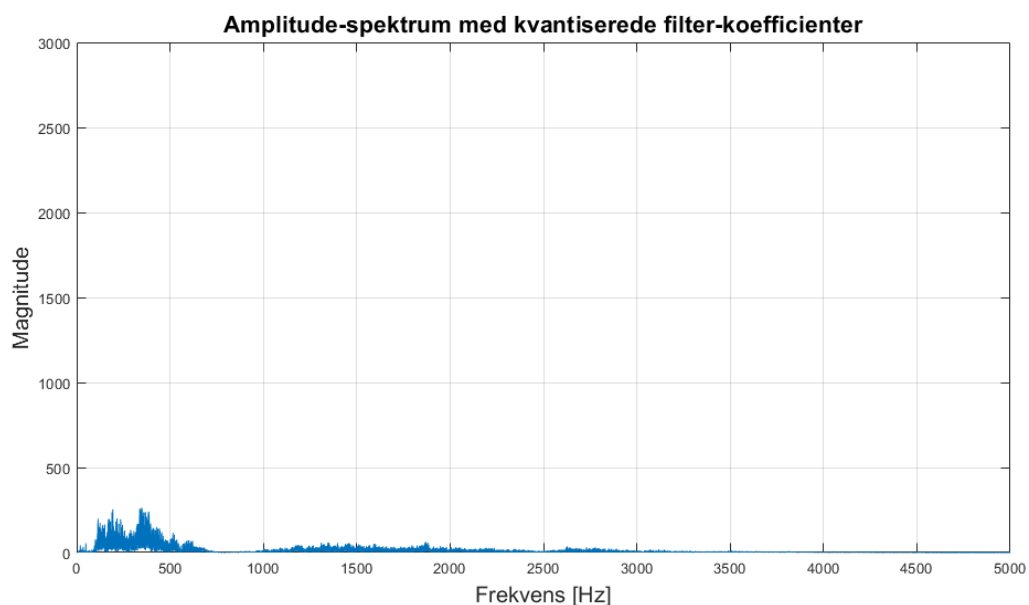
### Simuleret kvantisering af filter-koefficienter

Med matlab-funktionen på Figur 11 kan kvantisering af filter-koefficienterne også simuleres. Først kvantiseret filterkoefficienterne til 16 bit, dvs. fixed point Q1.15 værdier, da dette er samme grad af kvantisering, som vil opstå i blackfin-implementeringen, se filteret på Figur 13.



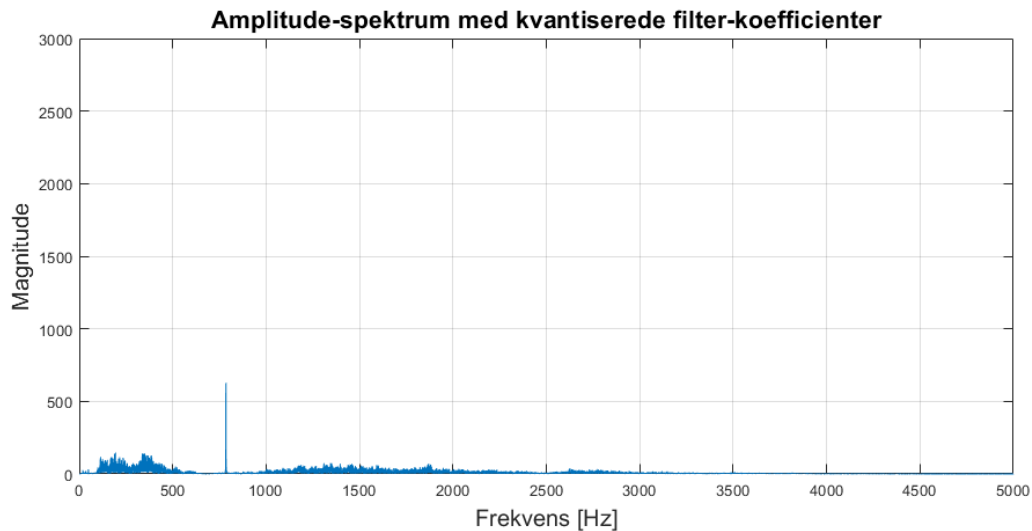
Figur 13 - Filter med kvantiserede koefficienter til 16 bit

Sammenligner man filteret på Figur 13 med filteret på Figur 9, har cut-frekvensen faktisk ikke ændret sig. Til gengæld er dæmpningen blevet lavere, men en dæmpning på -80dB er stadig rigeligt. Hvis det kvantiserede filter påføres lydsignalet, ses det at sinus-tonen stadig fjernes, og filteret har altså dermed stadig den ønskede effekt, hvorfor en implementering på blackfin vil fungere, se Figur 14.



Figur 14 - Frekvensspektre af lydsignalet filtreret med 16 bit kvantiserede filter-koefficienter

Hvis koefficienterne derimod kvantiseres til 8 bit værdier, så mister filteret sin effekt og fjerner ikke sinus-tonen, se Figur 15. Konklusionen er dog at kvantisering af filterkoefficienterne til 16 bit ikke har en betydelig effekt på filteret, dog ville et filter med dårligere opløsning være fatalt for formålet med filteret.



Figur 15 - Frekvensspektre af lydsginalet filtreret med 8 bit kvantiserede filter-koefficienter

### Simuleret kvantisering af produkterne i filteret

I en blackfin-implementering vil det ikke kun være koefficienterne i filteret som kvantiseres, men også selve produkterne i filteret. I koden på Figur 16 ses det hvordan filtreret påtrykkes, men hvor differensligningens produkter kvantiseres til 16 bit, dvs. Q1.15 format. Filter-koefficienterne er de 16 bit kvantiserede fra den foregående simulering.

```
bits = 16;

% Input dataen kvantiseres
audioKvant = quantize(X, bits);

% koefficienter quantized
b0q = quantize(b0,bits);
b1q = quantize(b1,bits);
b2q = quantize(b2,bits);

a0q = quantize(a0,bits);
a1q = quantize(a1,bits);
a2q = quantize(a2,bits);

% Array til at holde outputtet
audioProdQ = zeros(1,N)';

x0 = 0;
x1 = 0;
x2 = 0;
y1 = 0;
y2 = 0;

for i = 1:N

    % Input signal:
    x0 = audioKvant(i);

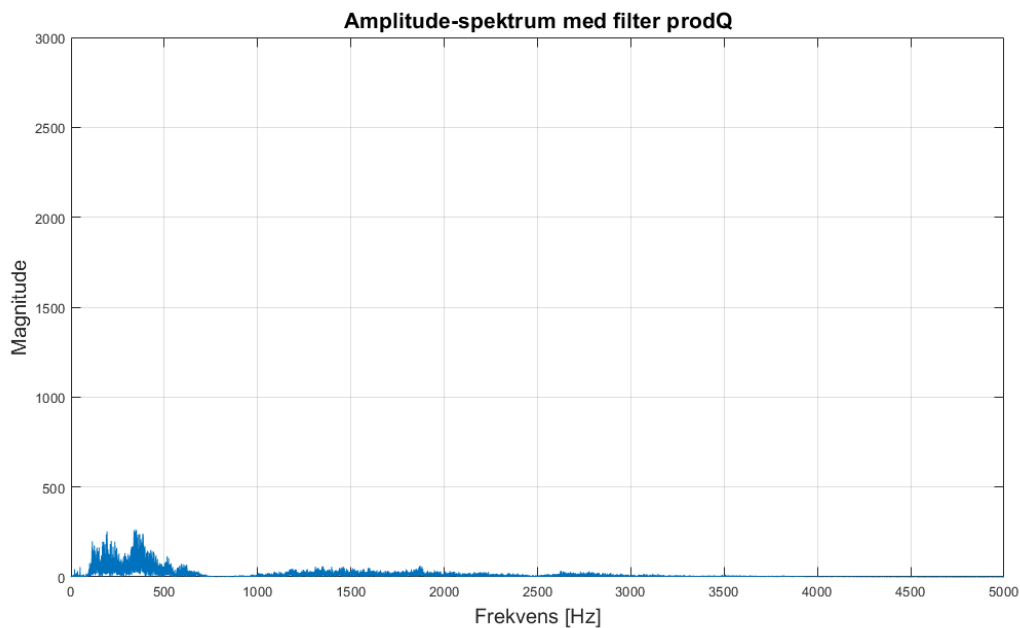
    % Differensligningen:
    audioProdQ(i) = quantize(b0q*x0,bits) + quantize(b1q*x1,bits) + quan-
tize(b2q*x2,bits) - quantize(a1q*y1,bits) - quantize(a2q*y2,bits);

    y2 = y1;
```

```
x2 = x1;  
x1 = x0;  
y1 = audioProdQ(i);  
end
```

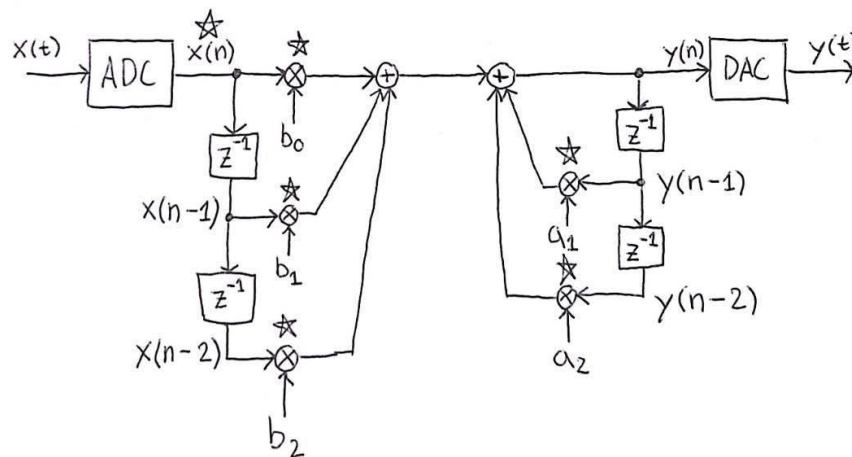
Figur 16 - Implementering af filter i matlab med produkt-kvantisering og med kvantiserede koefficienter

Efter kvantiseringen udføres der FFT og frekvensspektret for det filtrerede signal kan ses på Figur 17. Det ses tydeligt at kvantiseringen til 16 bit ikke har en hørbar effekt, hvilket også bekræftes ved at lytte til signalet med matlab-funktionen sound().



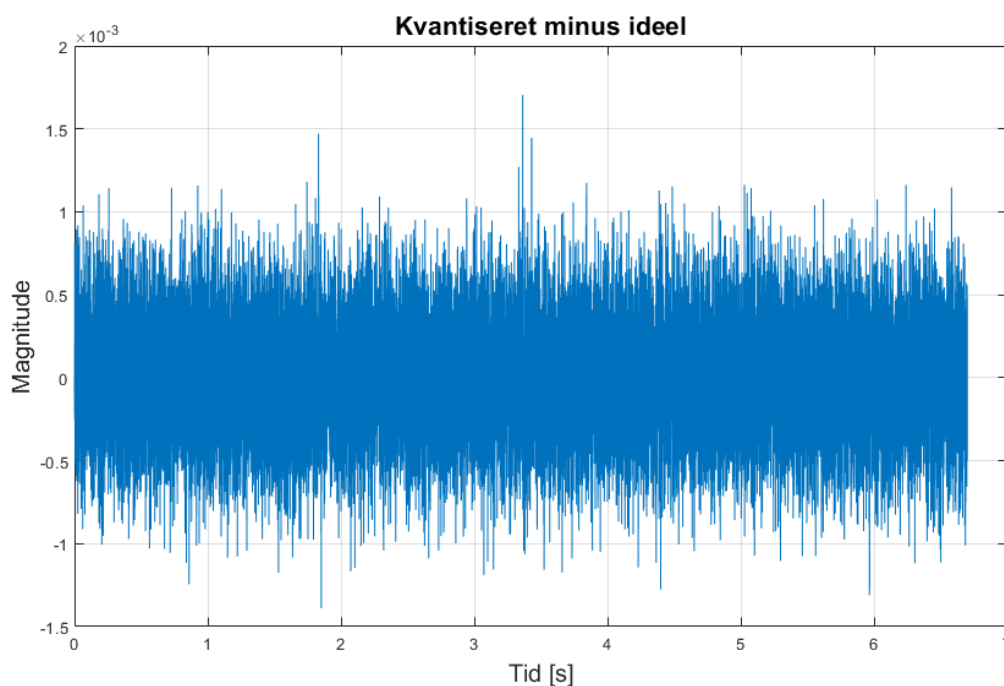
Figur 17 - Lydsignalet er her kvantiseret og påtrykket et filter, hvor koefficienterne er kvantiseret og produkterne i differensligningen er kvantiseret. Alt sammen til 16 bit, dvs. Q1.15 format.

På Figur 18 ses en signal-flow-graf over filteret. På grafen ses påtegnede stjerner, som symboliserer at der sker kvantisering. ADC'en udgør kvantiseringen af lydsignalet. Så er der tegnet stjerner over produkterne, da der her sker produktkvantisering, og så er koefficienterne også kvantiseret.



Figur 18 - Signal-flow-graf af notch-filteret. Der er påtegnet en stjerne de steder, hvor der sker kvantisering.

For at sammenligne det kvantiserede signal som er filtreret med det kvantiserede filter med det originale data filtreret med det ideelle filter på Figur 9, tages differensen mellem de to outputs og plottes i tidsdomænet, se Figur 19. Det er altså de fejl som kommer fra kvantiseringen til Q1.15 format, hvilket må siges at være småt. Lyd-kvaliteten har heller ikke ændret sig bemærkelsesværdigt. Det kan dermed konkluderes at en kvantisering til fixed point Q1.15 format stadig fjerner den uønskede sinus-tone og bevarer kvaliteten af signalet, således det stadig er hørbart.



Figur 19 - Det kvantiserede (Q1.15) filtrerede signal minus det ikke-kvantiserede signal filtreret med det ideelle filter.

## Opgave C - Implementering af notch-filter

Notch-filteret er implementeret som et IIR-filter med differensligningen:

$$y(n) \cdot a_0 = x(n) \cdot b_0 + x(n-1) \cdot b_1 + x(n-2) \cdot b_2 - a_1 \cdot y(n-1) - a_2 \cdot y(n-2)$$

Hvor koefficienterne er givet ved:

$$a_0 = 1$$

$$a_1 = 2 \cdot r \cdot \cos(\omega)$$

$$a_2 = -r^2$$

$$b_0 = 1$$

$$b_1 = -2 \cdot \cos(\omega)$$

$$b_2 = 1$$

Der er valgt en radius  $r = 0.95$ , og vinklen  $\omega$  er fundet ved  $\omega = 2 \cdot \pi \cdot f_c$ , hvor  $f_c = 785 \text{ Hz}$ . Dette giver følgende koefficienter: ( $a_1$  og  $a_2$  med omvendt fortegn)

```
//float a[3] = {1, -1.8900, 0.9025}; //Koefficienter fra matlab
//float b[3] = {1, -1.9895, 1};
```

Da koefficienter skal gemmes som 16-bit værdier i Q1.15 format, skales de alle med en faktor 2, således, at de ikke overfløder rangen  $[-1; 1 - 2^{-15}]$ . Dette medfører, at  $y(n)$  skal ganges med to, for at resultatet kommer til at passe. Efter skalering konverteres alle koefficienterne til hex i matlab vha. den funktionen myd2h. Dette giver:

```
short a[3] = {0x4000, 0x870A, 0x39C3}; //Koefficienter divideret med 2
short b[3] = {0x4000, 0x80AC, 0x4000}; //og konverteret til hex 1.15 format
```

Derudover initieres filterets delay-lines som shorts:

```
//Initiering af delay lines
short x1 = 0;
short x2 = 0;
short y1 = 0;
short y2 = 0;
```

Selve filter funktionen anvender den før beskrevne myMultiply-funktion, som sikrer returnering af en 16-bit værdi, til at gange samples og koefficienter sammen i differensligningen, som opskrives i koden i præcis samme form som vist ovenfor. Efter udregning sættes værdierne i delay linjen, hvorefter resultatet returneres. Se koden i Figur 20 nedenfor:

```
//float a[3] = {1, -1.8900, 0.9025}; //Koefficienter fra matlab
//float b[3] = {1, -1.9895, 1};

short a[3] = {0x4000, 0x870A, 0x39C3}; //Koefficienter divideret med 2
short b[3] = {0x4000, 0x80AC, 0x4000}; //og konverteret til hex 1.15 format

//Initiering af delay lines
short x1 = 0;
short x2 = 0;
short y1 = 0;
```

```
short y2 = 0;

short myNotchFilter(const short xn){

    static short res = 0;

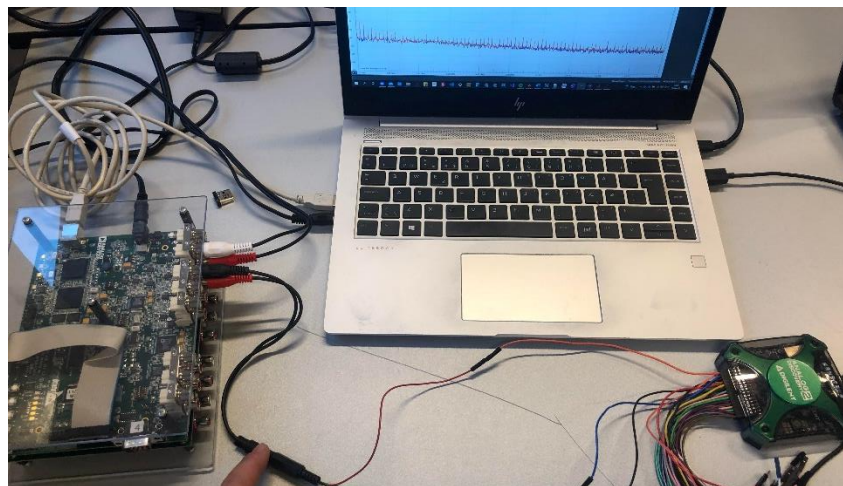
    //Differensligning
    res = myMultiply(xn,b[0]) + myMultiply(x1,b[1]) + myMultiply(x2,b[2])
        - myMultiply(y1,a[1]) - myMultiply(y2,a[2]);

    y2 = y1;        //Opsætning af Delay lines
    y1 = res/a[0];
    x2 = x1;
    x1 = xn;

    return y1;
}
```

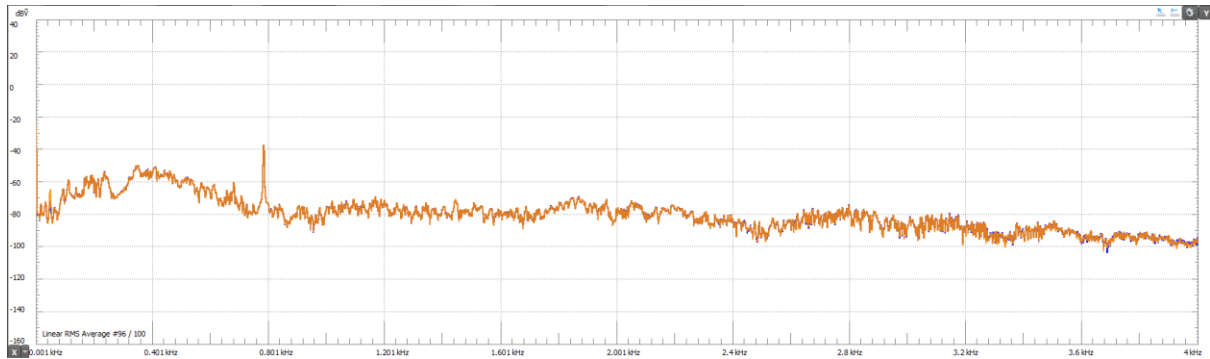
Figur 20 - Implementering af notch-filter

Implementering er testet ved fra PC at feede testsignaler ind i BlackFin-kittet, og måle på udgangen med Analog Discovery med filteret til og fra. På PC-en er der lavet fft-analyse, for at verificere at filteret virker som tiltænkt. Se opstillingen på Figur 21

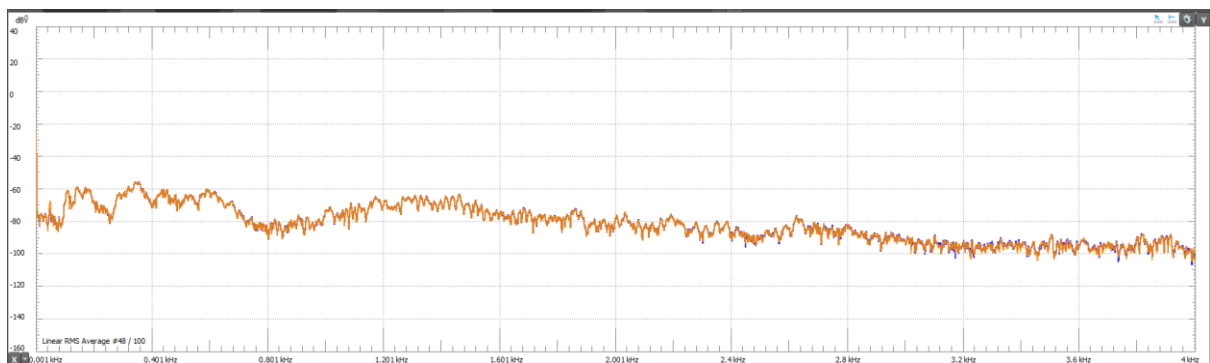


Figur 21 - Testopstilling til verificering af notch-filter implementering

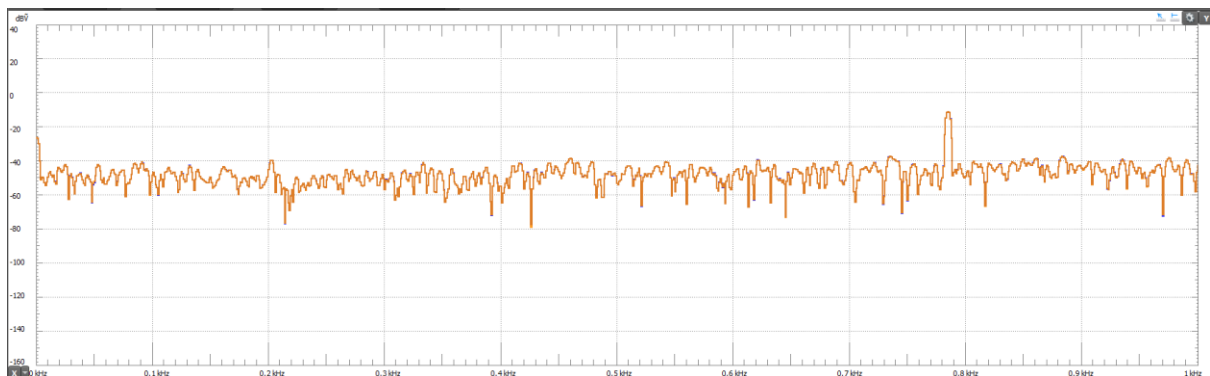
På næste side fremgår målingerne af dels et talesignal med sinustonen overlejret, og dels et hvidstøjssignal overlejret med samme sinustone. I begge tilfælde ses det, at frekvensen bliver fjernet fra signalet, hvormed det kan konkluderes, at filteret virker som forventet. Samme resultat er blevet bekræftet ved tilkobling af højttaler på udgangen, og aflytning af de to signaler med filteret til og fra.



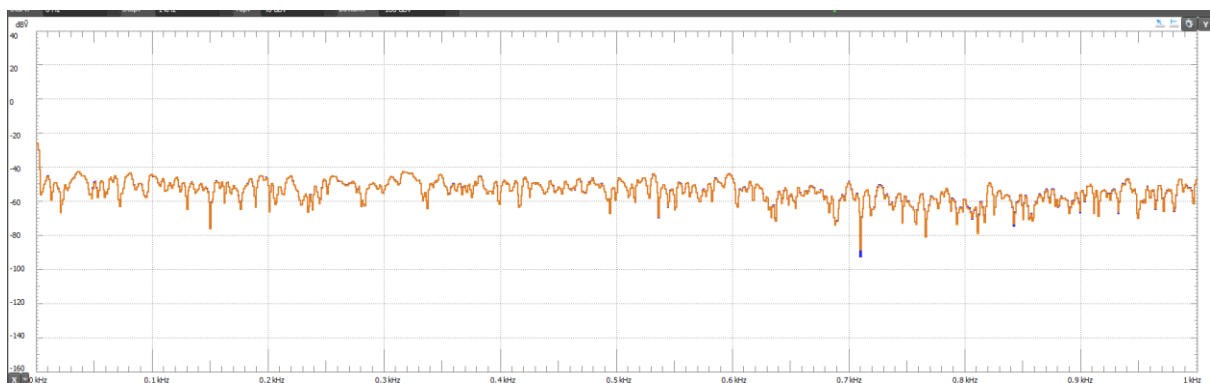
*Figur 22 - Tale + 785 Hz sinus , UDEN FILTER*



*Figur 23 - Tale + 785 Hz sinus, MED FILTER*



*Figur 24 - Hvid støj + 785 Hz sinus, UDEN FILTER*



*Figur 25 - Hvid støj + 785 Hz sinus, MED FILTER*